REDUCTION STRATEGIES IN REWRITING AND PROGRAMMING

Second International Workshop, WRS 2002
Copenhagen, Denmark, July 21, 2002

Proceedings

Bernhard Gramlich and Salvador Lucas, eds.

Technical Report E1852-2002-02

TECHNISCHE UNIVERSITÄT WIEN

Theory and Logic Group, Institute of Computer Languages (E185/2)
Vienna University of Technology, Favoritenstraße 9, A-1040 Wien, Austria

REDUCTION STRATEGIES IN REWRITING AND PROGRAMMING

Second International Workshop, WRS 2002
Copenhagen, Denmark, July 21, 2002

Proceedings

Bernhard Gramlich and Salvador Lucas, eds.

Technical Report E1852-2002-02

TECHNISCHE UNIVERSITÄT WIEN

Theory and Logic Group, Institute of Computer Languages (E185/2)
Vienna University of Technology, Favoritenstraße 9,A-1040 Wien,Austria

Title:

Reduction Strategies in Rewriting and Programming, Second International Workshop, WRS 2002, Copenhagen, Denmark, July 21, 2002, Proceedings

Edited by: Bernhard Gramlich and Salvador Lucas

Publisher:

Verlag Berger
Wiener Str. 80, A–3580 Horn, Austria

Published in: July 2002

ISBN: 3-85028-352-6

Preface

This volume contains the preliminary proceedings of the Second **International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2002**.[1] The workshop is held as part of FLoC 2002 in conjunction with RTA 2002 in Copenhagen, Denmark on July 21, 2002. WRS 2002 is the second edition in a series of workshops intended to stimulate and promote research and progress in this important field. WRS 2001[2] took place in Utrecht in conjunction with RTA 2001.

Background: Reduction strategies in rewriting and programming have attracted an increasing attention within the last years. New types of reduction strategies have been invented and investigated, and new results on rewriting / computation under particular strategies have been obtained. Research in this field ranges from primarily theoretical questions about reduction strategies to very practical application and implementation issues. The need for a deeper understanding of reduction strategies in rewriting and programming, both in theory and practice, is obvious, since they bridge the gap between unrestricted general rewriting (computation) and (more deterministic) rewriting with particular strategies (programming). Moreover, reduction strategies provide a natural way to go from operational principles (e.g., graph and term rewriting, narrowing, lambda-calculus) and semantics (e.g., normalization, computation of values, infinitary normalization, head-normalization) to implementations of programming languages.

WRS 2002 wants to provide a forum for the presentation and discussion of new ideas and results, recent developments, and new research directions, as well as of surveys on existing knowledge in this area. Furthermore we aim at fostering interaction and exchange between researchers and students actively working on such topics.

Topics of interest include, but are not restricted to:

- theoretical foundations for the definition and semantic description of reduction strategies,
- strategies in different frameworks (term rewriting, graph rewriting, infinitary rewriting, lambda calculi, higher order rewriting, conditional rewriting, rewriting with built-ins, narrowing, constraint solving, etc.) and their application in (equational, functional, functional-logic) programming (languages),

[1] See `http://www.dsic.upv.es/users/elp/WRS2002/`, `http://floc02.diku.dk/WRS/`.
[2] See `http://www.logic.at/wrs01/`.

- properties of reduction strategies / computations under strategies (e.g., completeness, computability, decidability, complexity, optimality, (hyper-) normalization, cofinality, fairness, perpetuality, context-freeness, neededness, laziness, eagerness, strictness),
- interrelations, combinations and applications of reduction under different strategies (e.g., equivalence conditions for fundamental properties like termination and confluence, applications in modularity analysis, connections between strategies of different frameworks, etc.),
- program analysis and other semantics-based optimization techniques dealing with reduction strategies,
- rewrite systems / tools / implementations with flexible / programmable strategies as essential concept / ingredient,
- specification of reduction strategies in (real) languages, and
- data structures and implementation techniques for reduction strategies.

The technical program of the workshop includes eight accepted papers, two invited talks by Aart Middeldorp (Tsukuba) and Vincent van Oostrom (Utrecht), and a final general discussion.

We wish to express our gratitude to all members of the program committee and to the external referees for their thorough work and care in reviewing and selecting papers. Furthermore, we would also like to thank all authors who submitted papers for their interest in the workshop. Finally, we are also grateful to the Local Organizing Committee of FLoC 2002 (especially to Nils Andersen, Niels H. Christensen, Andrzej Filinski, Klaus Ebbe Grue, Henning Makholm, Sebastian Skalbert) for a smooth collaboration regarding organizational issues.

July 2002
Bernhard Gramlich and Salvador Lucas
WRS 2002 Program Commitee Co-Chairs

Program Committee

Sergio Antoy	Portland State University
Roberto Di Cosmo	Université de Paris VII
Bernhard Gramlich (co-chair)	Technische Universität Wien
Michael Hanus	Christian-Albrechts-Universität zu Kiel
Claude Kirchner	LORIA & INRIA, Nancy
Paul Klint	CWI Amsterdam
Salvador Lucas (co-chair)	Universidad Politécnica de Valencia
Manfred Schmidt-Schauss	J. W. Goethe-Universität Frankfurt am Main
Yoshihito Toyama	Tohoku University

External Referees

María Alpuente

César Ferri

Matthias Mann

Huy Nguyen

Ricardo Peña

Table of Contents

Session 1

Approximations for Strategies and Termination (invited talk) 1
Aart Middeldorp

Regular Sets of Descendants by Leftmost Strategy 3
Pierre Réty and Julie Vuotto

Session 2

Term Rewriting with Type-safe Traversal Functions 21
Mark G.J. van den Brand, Paul Klint, and Jurgen J. Vinju

The Sketch of a Polymorphic Symphony 39
Ralf Lämmel

Strategies for Source-to-Source Constant Propagation 59
Karina Olmos and Eelco Visser

Session 3

Optimal Strategies in Higher-Order Rewriting 79
Vincent van Oostrom

An Abstract Böhm-normalization 81
John Glauert and Zurab Khasidashvili

Session 4

Operational Semantics for Lazy Functional Logic Programs 97
Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal

Pattern-driven Reduction in Haskell 113
William L. Harrison and Richard B. Kieburtz

Applying ELAN Strategies in Simulation Processors over Simple Architectures 127
Mauricio Ayala-Rincón, Rinaldi Maya Neto, Ricardo P. Jacobi, Carlos Llanos, and Reiner Hartenstein

Author Index 143

Approximations for Strategies and Termination

Aart Middeldorp [1]

Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba 305-8573, Japan

Abstract

The theorem of Huet and Lévy stating that for orthogonal rewrite systems (i) every reducible term contains a needed redex and (ii) repeated contraction of needed redexes results in a normal form if the term under consideration has a normal form, forms the basis of all results on optimal normalizing strategies for orthogonal rewrite systems. Unfortunately, needed redexes are not computable in general. Hence, in order to obtain a computable optimal strategy, we need to find (1) decidable approximations of neededness and (2) decidable classes of rewrite systems which ensure that every reducible term has a needed redex identified by (1).

In the talk we illustrate, based on the framework introduced in [2], how the use of approximations and tree automata techniques allows one to address these questions in a simple and elegant way.

We further show how the very same ideas can be used to improve [3] the dependency pair method of Arts and Giesl [1] for proving termination of rewrite systems automatically. More precisely, we show how approximations and tree automata techniques provide a better estimation of the dependency graph. This graph determines the ordering constraints that have to be solved in order to conclude termination. If time permits, we present a new estimation of the dependency graph that does not rely on computationally expensive tree automata techniques.

References

[1] Arts, Thomas, and Jürgen Giesl, *Termination of Term Rewriting Using Dependency Pairs*, Theoretical Computer Science **236** (2000), 133–178.

[2] Durand, Irène, and Aart Middeldorp, *Decidable Decidable Call by Need Computations in Term Rewriting (Extended Abstract)*, Proceedings of the 14th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence **1249** (1997), 4–18.

[1] Email: `ami@is.tsukuba.ac.jp`.

[3] Aart Middeldorp, *Approximating Dependency Graphs using Tree Automata Techniques*, Proceedings of the International Joint Conference on Automated Reasoning, Lecture Notes in Artificial Intelligence **2083** (2001), 593–610.

Regular Sets of Descendants by Leftmost Strategy

Pierre Réty [1] Julie Vuotto [2]

LIFO - Université d'Orléans
B.P. 6759, 45067 Orléans cedex 2, France

Abstract

For a constructor-based rewrite system R, a regular set of ground terms E, and assuming some additional restrictions, we build a finite tree automaton that recognizes the descendants of E, i.e. the terms issued from E by rewriting, according to leftmost strategy.

Keywords : term rewriting, strategy, tree automaton.

1 Introduction

Reduction strategies in rewriting and programming have drawn an increasing attention within the last years, and matter both from a theoretical point of view, if the computation result is not unique, and from a practical point of view, for termination and efficiency. For a leftmost-strategy $left$, expressing by a finite tree automaton the $left$-descendants $R^*_{left}(E)$ of E, can help to study $left$-strategy: in particular it allows to decide $left$-reachability since $t_1 \,{}^{left}\!\!\rightarrow^* t_2 \iff t_2 \in R^*_{left}(\{t_1\})$, and $left$-joinability since $t_1 \overset{left}{\downarrow} t_2 \iff R^*_{left}(\{t_1\}) \cap R^*_{left}(\{t_2\}) \neq \emptyset$. More generally, it can help with the static analysis of rewrite programs, and by extension, of functional programs. Computations of innermost, innermost-leftmost and outermost descendants have been investigated in [3] by the authors.

We build a finite automaton that can express the sets of leftmost descendants of E with respect to a constructor-based rewrite system, assuming:

(i) E is the set of ground constructor-instances (also called data-instances) of a given linear term t (i.e. $E = \{t\theta\}$).

(ii) Every rewrite rule is linear (both sides).

---

[1] Email: `rety@lifo.univ-orleans.fr`,
www: `http://www.univ-orleans.fr/SCIENCES/LIFO/Members/rety/`

[2] Email: `vuotto@lifo.univ-orleans.fr`

(iii) In right-hand-sides, there are no nested defined-functions, and arguments of defined-functions are either variables or ground terms.

(iv) Permutative rewrite rules are forbidden.

(v) Every rewrite rule is variable-preserving. By transforming R, Restriction v can be weakened into restriction v': every rewrite rule is left-variable-preserving[3].

It is shown in [2] that if any restriction among 1, 2, 3 is not satisfied, then the set of descendants without a strategy is not regular.

Compared to the innermost-leftmost descendants of [3], we assume more restrictions (iv, v). This is due to the fact that computing leftmost descendants is more difficult than computing only innermost-leftmost ones. However, unlike [3], rhs's containing several defined-functions are allowed, thanks to the new result of section 3.

The paper is organized as follows. Section 3 gives the computation of the language $R_p^{*\lhd}(\{t\theta\})$ (i.e. terms obtained after rewriting eventually at position p of t plus possibly at functions positions issued from rhs's, according to a leftmost strategy). Leftmost descendants are computed in section 4 (we apply the previous language for every p position of t, by checking out before that sub-terms occurring on the left of p be normalized by intersection with an automaton that recognized irreducible terms). Now, let us see, in the following section preliminaries notions. Subsection 2.1 (usual notions) may be skipped by the reader.

2 Preliminaries

2.1 Usual notions: term rewriting and tree automata

Let C be a finite set of *constructors* and F be a finite set of *defined-function symbols* (*functions* in a shortened form). For $c \in C \cup F$, $ar(c)$ is the arity of c. *Terms* are denoted by letters s, t. A *data-term* is a *ground* term (i.e. without variables) that contains only constructors. T_C is the set of data-terms, and $T_{C\cup F}$ the set of ground terms that contains constructors and functions. For a term t, $Var(t)$ is the set of variables appearing in t, $Pos(t)$ is the set of *positions* of t, $\overline{Pos}(t)$ is the set of non-variable positions of t, $PosF(t)$ is the set of defined-function positions of t. $PosVar(t)$ is the set of variable positions of t. For $p \in Pos(t)$, $t|_p$ is the subterm of t at position p, $t(p)$ is the top symbol of $t|_p$, and $t[t']_p$ denotes the subterm replacement. For positions p, p', $p \geq p'$ means that p is located below p', i.e. $p = p'.v$ for some position v, whereas $p||p'$ means that p and p' are incomparable, i.e. $\neg(p \geq p') \wedge \neg(p' \geq p)$. The term t contains *nested functions* if there exist $p, p' \in PosF(t)$ s.t. $p > p'$. The domain $dom(\theta)$ of a substitution θ is the set of variables x s.t. $x\theta \neq x$.

A *rewrite rule* is an oriented pair of terms, written $l \to r$. A *rewrite system* R is a finite set of rewrite rules. *lhs* stands for left-hand-side, *rhs*

[3] if $x \in Var(l) \cap Var(r)$ and $y \in Var(l) - Var(r)$, then x occurs in l on the left of y.

for right-hand-side. R is *constructor-based* if every lhs l of R is of the form $l = f(t_1, \ldots, t_n)$ where $f \in F$ and $t_1, \ldots, t_n$ do not contain any functions. The rewrite relation $\rightarrow_R$ is defined as follows: $t \rightarrow_R t'$ if there exist $p \in Pos(t)$, a rule $l \rightarrow r \in R$, and a substitution θ s.t. $t|_p = l\theta$ and $t' = t[r\theta]_p$ (also denoted by $t \rightarrow_{[p, l \rightarrow r, \theta]} t'$). $\rightarrow_R^*$ denotes the reflexive-transitive closure of $\rightarrow_R$. t is *irreducible* if $\neg(\exists t' \mid t \rightarrow_R t')$. $t \rightarrow_{[p]} t'$ is *leftmost* if $\forall v$ occurring strictly on the left of p, $t|_v$ is irreducible.

A (bottom-up) finite tree *automaton* is a quadruple $\mathcal{A} = (C \cup F, Q, Q_f, \Delta)$ where $Q_f \subseteq Q$ are sets of states and Δ is a set of *transitions* of the form $c(q_1, \ldots, q_n) \rightarrow q$ where $c \in C \cup F$ and $q_1, \ldots, q_n, q \in Q$, or of the form $q_1 \rightarrow q$ (*empty transition*). Sets of *states* are denoted by letters Q, S, D, and states by q, s, d. $\rightarrow_\Delta$ (also denoted $\rightarrow_{\mathcal{A}}$) is the rewrite relation induced by Δ. A ground term t is *recognized* by $\mathcal{A}$ into q if $t \rightarrow_\Delta^* q$. $L(\mathcal{A})$ is the set of terms recognized by $\mathcal{A}$ into any states of Q_f. A derivation $t \rightarrow_\Delta^* q$ where $q \in Q_f$ is called a *successful run* on t. The states of Q_f are called *final states*. A Q*-substitution* σ is a substitution s.t. $\forall x \in dom(\sigma)$, $x\sigma \in Q$. A set E of ground terms is regular if there exists a finite automaton $\mathcal{A}$ s.t. $E = L(\mathcal{A})$.

2.2 Nesting Automata and Discrimination

Intuitively, the automaton $\mathcal{A}$ discriminates position p into state q means that along every successful run on $t \in L(\mathcal{A})$, $t|_p$ (and only this subterm) is recognized into q. This property allows to modify the behavior of $\mathcal{A}$ below position p without modifying the other positions, by replacing $\mathcal{A}|_p$ by another automaton $\mathcal{A}'$. See [2] for proofs of Lemmas 2.2 and 2.4, and [3] for the proof of Lemma 2.5.

Definition 2.1 The automaton $\mathcal{A} = (C \cup F, Q, Q_f, \Delta)$ *discriminates* the position p into the state q if

- $L(\mathcal{A}) \neq \emptyset$,
- and $\forall t \in L(\mathcal{A})$, $p \in Pos(t)$,
- and for each successful derivation $t \rightarrow_\Delta^* t[q']_{p'} \rightarrow_\Delta^* q_f$ where $q_f \in Q_f$, we have
  - $q' \rightarrow_\Delta^* q$ (i.e. by empty transitions) if $p' = p$,
  - $q' \neq q$ otherwise.

In this case we define the automaton $\mathcal{A}|_p = (C \cup F, Q, \{q\}, \Delta)$.

Lemma 2.2 *Then* $L(\mathcal{A}|_p) = \{t|_p \mid t \in L(\mathcal{A})\}$.

Definition 2.3 Let $\mathcal{A} = (C \cup F, Q, Q_f, \Delta)$ be an automaton that discriminates the position p into the state q, and let $\mathcal{A}' = (C \cup F, Q', Q'_f, \Delta')$ s.t. $Q \cap Q' = \emptyset$. We define

$$\mathcal{A}[\mathcal{A}']_p = (C \cup F,\, Q \cup Q',\, Q_f,\; \Delta \backslash \{l \rightarrow r \mid l \rightarrow r \in \Delta \wedge r = q\} \\ \cup \Delta' \cup \{q'_f \rightarrow q \mid q'_f \in Q'_f\})$$

Lemma 2.4 *$L(\mathcal{A}[\mathcal{A}']_p) = \{t[t']_p \mid t \in L(\mathcal{A}), t' \in L(\mathcal{A}')\}$, and $\mathcal{A}[\mathcal{A}']_p$ still discriminates p into q. Moreover, if $\mathcal{A}$ discriminates another position p' s.t. $p' \not\geq p$, into the state q', then $\mathcal{A}[\mathcal{A}']_p$ still discriminates p' into q'.*

Lemma 2.5 *Let $\mathcal{A}, \mathcal{B}$ be automata, and let $\mathcal{A} \cap \mathcal{B}$ be the classical automaton used to recognize intersection, whose states are pairs of states of $\mathcal{A}$ and $\mathcal{B}$. If $\mathcal{A}$ discriminates p into $q_{\mathcal{A}}$, $\mathcal{B}$ discriminates p into $q_{\mathcal{B}}$, and $L(\mathcal{A}) \cap L(\mathcal{B}) \neq \emptyset$, then $\mathcal{A} \cap \mathcal{B}$ discriminates p into $(q_{\mathcal{A}}, q_{\mathcal{B}})$.*

2.3 Positions and Antecedents

Given a term t, we define :

Definition 2.6 Let $p \in Pos(t)$. $Succ(p)$ are the nearest function positions below p :

$$Succ(p) = \{p' \in PosF(t) \mid p' > p \ \wedge \ \forall q \in Pos(t) \ (p < q < p' \Rightarrow q \notin PosF(t))\}$$

Definition 2.7 Let $p, p' \in Pos(t)$. $p \lhd p'$ means that p occurs strictly on the left of p', i.e. $p = u.i.v$, $p' = u.i'.v'$, where $i, i' \in \mathbb{N}$ and $i < i'$.

Definition 2.8 Let $t \rightarrow_{[q,l \rightarrow r]} t'$ be a rewrite step.
Let $v' \in Pos(t')$. v is an antecedent of v' in t (denoted $ant(v', t)$) through this step if:

(i) $v \in Pos(t)$

(ii)
- $v' \lhd q$ *and* $v = v'$ or,
- $\exists p' \in PosVar(r)$ *with* $r|_{p'} = x$ *s.t.* $v' = q.p'.w$ *and* $v = q.p''.w$ *where* p'' is a position of x in l.

Remark : Since lhs's are linear, the antecedent (if exists) is unique.

Definition 2.9 Consider the following derivation:

$$t_0 \rightarrow_{[p_0, l_0 \rightarrow r_0]} t_1 \rightarrow \ldots t_n \rightarrow_{[p_n, l_n \rightarrow r_n]} t_{n+1} \qquad (1)$$

$v_0 \in Pos(t_0)$ is an antecedent of $v_{n+1} \in Pos(t_{n+1})$ through this derivation if $\exists v_1 \in pos(t_1), \ldots, v_n \in Pos(t_n)$ s.t. $\forall i \in \{0, ..., n\}$, v_i is an antecedent of v_{i+1} through step $t_i \rightarrow t_{i+1}$.

2.4 Descendants and irreducible ground terms

t' is a *descendant* of t if $t \rightarrow_R^* t'$. t' is a *normal-form* of t if $t \rightarrow_R^* t'$ and t' is irreducible. If E is a set of ground terms, $R^*(E)$ denotes the set of descendants of elements of E. $IRR(R)$ denotes the set of irreducible ground terms. $R^*_{left}(E)$ denotes the set of descendants of E, according to a leftmost strategy.

Definition 2.10 $t \rightarrow^+_{[p, rhs's]} t'$ means that t' is obtained by rewriting t at position p, plus possibly at positions coming from the rhs's.

Formally, there exist some intermediate terms $t_1, \ldots, t_n$ and some sets of positions $P(t), P(t_1), \ldots, P(t_n)$ s.t.

$$t = t_0 \rightarrow_{[p_0, l_0 \rightarrow r_0]} t_1 \rightarrow_{[p_1, l_1 \rightarrow r_1]} \ldots \rightarrow_{[p_{n-1}, l_{n-1} \rightarrow r_{n-1}]} t_n \rightarrow_{[p_n, l_n \rightarrow r_n]} t_{n+1} = t'$$

where

- $p_0 = p$ and $P(t) = \{p\}$,
- $\forall j,\ p_j \in P(t_j)$,
- $\forall j,\ P(t_{j+1}) = P(t_j) \backslash \{p' \mid p' \geq p_j\} \cup \{p_j.w \mid w \in PosF(r_j)\}$.

Remark : $P(t_j)$ contains only function positions. Since there are no nested functions in rhs's, $p, p' \in P(t_j)$ implies $p \| p'$.

Definition 2.11 Given a language E and a position p, we define $R_p^{*\lhd}(E)$ as follows

$$R_p^{*\lhd}(E) = E \cup \{t' \mid \exists t \in E, t \rightarrow^+_{[p, rhs's]} t' \textit{ by leftmost rewriting}\}$$

Example 2.12 $R = \{f(x) \rightarrow s(x),\ g(x,y) \rightarrow c(h(x), f(y)),\ h(x) \rightarrow f(x)\}$ $R_1^{*\lhd}(\{f(g(a,b))\}) = \{f(g(a,b))\} \cup \{f(c(h(a), f(b)))\} \cup \{f(c(f(a), f(b)))\} \cup \{f(c(s(a), f(b)))\} \cup \{f(c(s(a), s(b)))\}$.

Definition 2.13 Let $IRR_p(R) = \{s \in T_{C \cup F} \mid p \in Pos(s) \wedge s|_p$ is irreducible$\}$.

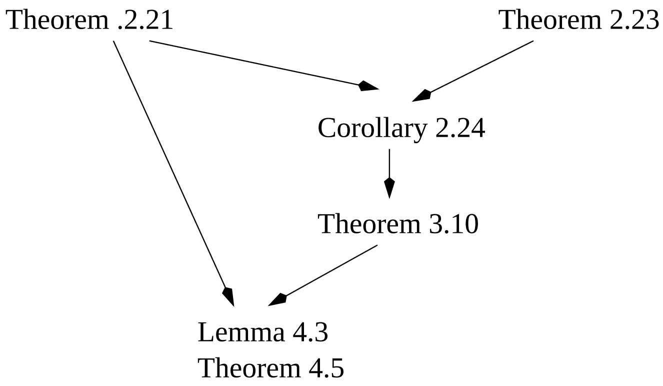
Theorem 2.14 *Let t be a term, and $p \in \overline{Pos}(t)$. $IRR_p(R)$ is a regular language and is recognized by an automaton that discriminates every position $p' \in \overline{Pos}(t)$ s.t. $p' \not\geq p$.*

Proof. See [3].
Let $\mathcal{A}_{irr}$ an automaton that recognized $IRR_\epsilon(R)$. In the following, we will call q^{irr} the accepting state of $\mathcal{A}_{irr}$. □

2.5 Commutation of Rewriting and Left-Basic Derivation

In a leftmost strategy, before rewriting at some position p, the subterms occurring on the left of p must be rewritten into normal forms, by leftmost derivations too. However, in order to avoid a loop when building the automaton, we normalize these subterms by derivations without strategy, and we show that the normal forms obtained by leftmost derivations and by arbitrary derivations are the same. For this, we show that an arbitrary derivation can always be transformed into a left-basic derivation, and that a left-basic derivation leading to a normal form is leftmost.

The following figure shows the links between lemmas and theorems. Theorem 4.4 corresponds to the final result.



By the two following Lemmas, we see how to commute a derivation (in two steps, next in several steps).

Lemma 2.15 *Let R be a linear TRS and s, t, u be terms. If*

$$s \to_{[p,g\to d]} t \to_{[q,l\to r]} u \qquad (1)$$

is a derivation in two steps s.t. q admits an antecedant in s denoted by p_0. Then, (1) can be commuted into:

$$s \to_{[p_0,l\to r]} t' \to_{[p,g\to d]} u' \qquad (2)$$

Now, let us suppose restrictions v and iv. Then, if (1) is leftmost, (2) is leftmost.
Remark : *R being linear, $g \to d$ is linear and so d is linear, consequently $u' = u$: The last term of derivation is preserved by commutation. Moreover, in (2), p does not have an antecedant in s.*

Lemma 2.16 *Let R be a linear TRS. If*

$$t_0 \to_{[p_0,l_0\to r_0]} t_1 \to \ldots t_{n-1} \to_{[pn-1,l_{n-1}\to r_{n-1}]} t_n \to_{[p_n,l_n\to r_n]} t_{n+1} \qquad (1)$$

is a derivation s.t. p_n admits an antecedant q_{n-1} in t_0. Then, (1) can be commuted in:

$$t_0 \to_{[q_{n-1},l_n\to r_n]} t'_1 \to_{[p_0,l_0\to r_0]} \ldots t'_n \to_{[p_{n-1},l_{n-1}\to r_{n-1}]} t'_{n+1} \qquad (2)$$

with $t'_{n+1} = t_{n+1}$ and p_0 has no antecedant in t_0.
Now, let us suppose restrictions v and iv. Then, if (1) is leftmost, (2) is leftmost too.

Now, let us define the notion of left-basic derivation.

Definition 2.17 Let the following derivation:

$$t_0 \to_{[p_0,l_0\to r_0]} t_1 \to \ldots t_n \to_{[p_n,l_n\to r_n]} t_{n+1} \qquad (1)$$

This derivation is said left-basic if : there exist sets of positions $B(t_0)$, $B(t_1), \ldots$, $B(t_n)$ for the terms $t_0, t_1, \ldots, t_n$ s.t.

- $B(t_0) = PosF(t_0)$
- $\forall j,\ p_j \in B(t_j)$

- $\forall j,\ B(t_{j+1}) = \{p' \mid p' \leq p_j\} \cup \{p_j.v \mid v \in PosF(r_j)\} \cup \{p' \mid p_j \lhd p'\}$

Note that leftmost-innermost implies left-basic. The converse is false in the general case.

Counterexample 2.18
Let $R = \{f(x) \to s(f(x)),\ h(x,y) \to c(f(x), s(g(y))),\ g(x) \to x\}$ *and* $E = h(s^*(a), s^*(a))$. *Consider the following derivation :*

$$E \to_{[\epsilon]} c(f(s^*(a)), s(g(s^*(a)))) \to_{[2.1]} c(f(s^*(a)), s(s^*(a)))$$

This derivation is left-basic but is not leftmost-innermost because of the rewrite step at position 2.1 *whereas position* 1 *is not normalized.*

Lemma 2.19 *Let the following derivation:*

$$t_0 \to_{[p_0, l_0 \to r_0]} t_1 \to \dots t_n \to_{[p_n, l_n \to r_n]} t_{n+1} \qquad (1)$$

then, (1) *is left-basic if and only if* $\forall i \in \{1, \dots, n\}$, p_i *has no antecedent in* t_{i-1}.

Lemma 2.20 *Let the following left-basic derivation:*

$$t_0 \to_{[p_0, l_0 \to r_0]} t_1 \to \dots t_n \qquad (1)$$

followed by the non-left-basic step:

$$t_n \to_{[p_n, l_n \to r_n]} t_{n+1}$$

Let us remark that p_n *admits an antecedent in* t_{n-1}. *Let* $j \leq n$ *the smaller integer s.t.* p_n *admits an antecedent in* t_j.
Then, the following derivation obtained by commutation:

$$t_0 \to_{[p_0, l_0 \to r_0]} t_1 \to \dots t_j \to_{[q, l_n \to r_n]} t'_{j+1} \to_{[p_j, l_j \to r_j]} t'_{j+2} \to \dots$$
$$\dots \to t'_n \to_{[p_{n-1}, l_{n-1} \to r_{n-1}]} t_{n+1} \qquad (2)$$

is left-basic.

Theorem 2.21 *Let* R *be a linear TRS. If* $t_0 \to^* t_n$ (1) *then,* $t_0 \to^* t_n$ (2) *by a left-basic derivation. Now, let us suppose restrictions v and iv. Then, if (1) is leftmost, (2) is leftmost too.*

Lemma 2.22 *Let* R *a given constructor-based TRS and assume restriction v. Let the following left-basic derivation:*

$$t_0 \to_{[p_0, l_0 \to r_0]} t_1 \to \dots t_n \to_{[p_n, l_n \to r_n]} t_{n+1} \qquad (1)$$

and let $j \in \{0, \dots, n\}$. *If* $\exists\ p' \in \overline{Pos}(t_j) - B(t_j)$ *s.t.* $t_j(p') \in \mathcal{F}$, *then* $\forall k > j$, $\exists p'_k \in \overline{Pos}(t_k) - B(t_k)$ *s.t.* $t_k|_{p'_k} = t_j|_{p'}$.

Theorem 2.23 *Let* R *a given constructor-based TRS and assume restriction v. Let the following left-basic derivation:*

$$t_0 \to^* t_{n+1}\ s.t.\ t_{n+1} = t_o\downarrow \qquad (1)$$

Then, (1) is leftmost.

Corollary 2.24 *Let R a given constructor-based TRS and assume restrictions ii and v. The normal forms of a given term t obtained by a leftmost rewrite strategy are the same as those obtained without rewrite strategy.*

Proof. Obviously, it is a consequence of Theorem 2.21 and of Theorem 2.23.□

2.6 Starting Automaton

Let us define the initial automaton, i.e. the automaton that recognizes the set of data-instances of a given linear term t.

Definition 2.25 We define the automaton $\mathcal{A}_{data}$ that recognizes the set of data-terms T(C):
$\mathcal{A}_{data} = (C, Q_{data}, Q_{data_f}, \Delta_{data})$ where $Q_{data} = Q_{data_f} = \{q_{data}\}$ and $\Delta_{data} = \{c(q_{data}, \ldots, q_{data}) \to q_{data} \mid c \in C\}$.

Given a linear term t, we define the automaton $\mathcal{A}_{t\theta}$ that recognizes the set of data-instances of t : $\mathcal{A}_{t\theta} = (C \cup F, Q_{t\theta}, Q_{t\theta_f}, \Delta_{t\theta})$ where

$$Q_{t\theta} = \{q^p \mid p \in \overline{Pos}(t)\} \cup \{q_{data}\}$$

$$Q_{t\theta_f} = \{q^\epsilon\} \text{ } (q_{data} \text{ if t is a variable})$$

$$\Delta_{t\theta} = \left\{ t(p)(s_1, \ldots, s_n) \to q^p \mid p \in \overline{Pos}(t), \; s_i = \left| \begin{array}{l} q_{data} \text{ if } t|_{p.i} \text{ is a variable} \\ q^{p.i} \text{ otherwise} \end{array} \right. \right\}$$
$$\cup \Delta_{data}$$

Note that $\mathcal{A}_{t\theta}$ discriminates each position p $\in \overline{Pos}(t)$ into q^p. On the other hand, $\mathcal{A}_{t\theta}$ is not deterministic, as soon as there is $p \in \overline{Pos}(t)$ s.t. $t|_p$ is a constructor-term. Indeed for any data-instance $t|_p\theta$, $t|_p\theta \to^*_{[\Delta_{t\theta}]} q^p$ and $t|_p\theta \to^*_{[\Delta_{t\theta}]} q_{data}$.

3 Recognizing $R_p^{*\triangleleft}(E)$

Notions introduced in this section are illustrated by example 3.11.

It may occur that the matches used in rewrite steps instantiate the variables by languages not recognized into states of $\mathcal{A}_E$. I.e the instances are not always (sub)terms of E. Let us see the following example:

Example 3.1 Let $R = \{g(x) \overset{r1}{\to} h(x, b),\ h(x, y) \overset{r2}{\to} c(x, y)\}$ and E be the set of data-instances of $t = g(a)$.
$\mathcal{A}_{t\theta}$ can be summarized by writing : $\overset{q^\epsilon}{g}\ (\overset{q^1}{a})$ (which means that $g(a) \to_{\Delta_{t\theta}} g(q^1) \to_{\Delta_{t\theta}} q^\epsilon$). The rewrite steps issued from E are $g(a) \to_{[\epsilon, r1, x/a]} h(a, b) \to_{[\epsilon, r2, x/a\ y/b]} c(a, b)$. Unfortunately, $Q_{t\theta} = \{q^\epsilon,\ q^1\}$ and the language recognized into q^ϵ (resp. q^1) is $g(a)$ (resp. a).Thus, we do not have any states that can recognize $\{b\}$. This comes from the fact that $\{b\}$ is provided by the rhs $r1$. Therefore, we need to encode $\{b\}$ by additional states.
So, we give the following definition.

Definition 3.2 The non-variable arguments of functions in rhs's are encoded by the set of states Q_{arg} and the set of transitions Δ_{arg} as defined below:

$$Q_{arg} = \{q^{i,p} \mid \ l_i \to r_i \in R,\ p \in Arg(r_i)\}$$
$$\Delta_{arg} = \{r_i(p)(q^{i,p.1}, \dots, q^{i,p.n}) \to q^{i,p} \mid \ q^{i,p} \in Q_{arg}\}$$

where $Arg(r_i)$ are the non-variable argument positions in r_i, i.e.

$$Arg(r_i) = \{p \in \overline{Pos}(r_i) \mid \ \exists p_{fct} \in PosF(r_i),\, p > p_{fct}\}$$

Now, we define how to encode a version of each instantiated rhs.
Let $\mathcal{A} = (C \cup F, Q, Q_f, \Delta)$ an automaton that discriminates the position p into the state q, s.t. $Q \cap Q_{arg} = \emptyset$ (notation used in the continuation of this section). Let $Q' = Q \cup Q_{arg}$. We use states of the form d^p_σ where σ is a Q'-substitution, because rhs's may contain variables.

Definition 3.3 The rhs's of rewrite rules are encoded by the sets of states Q_{arg} and

$$D = \{d^{i,p}_\sigma \mid \ l_i \to r_i \in R,\ p \in Pos(r_i) \backslash Arg(r_i),$$
$$\sigma \text{ is a } Q'\text{-substitution s.t. } dom(\sigma) = Var(r_i|_p)\}$$

and the set of transitions

$$\begin{aligned}\Delta_d = \{r_i(p)(X_1, \dots, X_n) \to d^{i,p}_{\sigma_1 \cup \dots \cup \sigma_n} \mid \ & l_i \to r_i \in R, \\ & p \in Pos(r_i) \backslash Arg(r_i),\ r_i(p) \in C \\ & \forall j,\ \sigma_j \text{ is any } Q'\text{-substitution s.t. } dom(\sigma_j) = Var(r_i|_{p.j}), \\ & \text{where } \forall j,\ X_j = \left| \begin{array}{l} x\sigma_j \mid \ r_i(p.j) \text{ is any variable } x \\ d^{i,p.j}_{\sigma_j} \text{ otherwise} \end{array} \right. \} \\ \cup \{r_i(p)(X_1, \dots, X_n) \to d^{i,p}_\sigma \mid \ & l_i \to r_i \in R,\ p \in PosF(r_i), \\ & \sigma \text{ is any } Q'\text{-substitution s.t. } dom(\sigma) = Var(r_i|_p) \\ & \text{where } \forall j,\ X_j = \left| \begin{array}{l} x\sigma \mid \ r_i(p.j) \text{ is any variable } x \\ q^{i,p.j} \in Q_{arg} \text{ otherwise} \end{array} \right. \} \\ \cup \{x\sigma \to d^{i,\epsilon}_\sigma \mid & l_i \to r_i \in R,\ r_i \text{ is any variable } x, \\ & \sigma \text{ is any } Q'\text{-substitution s.t. } dom(\sigma) = \{x\}\}\end{aligned}$$

Thus, $r_i\sigma$ and only it, is recognized into the state $d^{i,\epsilon}_\sigma$.

Definition 3.4 Let $D^{sat} and \Delta^{sat}_d$ the set of states and transitions obtained as in Definition 3.3 by replacing each state $d^{i,p}_\sigma \in D$ by $d^{i,p}_{sat,\sigma}$.

The goal of these two similar encodings of rhs's is to recognize the instances of rhs's thanks to (D, Δ_d), and theirs descendants thanks to $(D^{sat}, \Delta^{sat}_d)$ and Δ^{sat} generated by the saturation process defined below:

Definition 3.5 (*saturation*)
Let Δ^{sat} be the set of transitions added in the following way: whenever there are $l_i \to r_i \in R$, a $(Q \cup Q_{arg})$-substitution σ s.t. $dom(\sigma) = Var(l_i) \cup Var(r_i)$ and $l_i\sigma \to^*_{\Delta_{t\theta} \cup \Delta_{arg} \cup \Delta^{sat}_d} q'$ where $q' \in \{q\} \cup D^{sat}$, add the transition $d^{i,\epsilon}_{sat,\sigma|_{Var(r_i)}} \to q'$.

Notation : $\Delta^{sat*}_d = \Delta^{sat}_d \cup \Delta^{sat}$.
Remark : Let us note $\mathcal{B}' = (C \cup F, Q \cup Q_{arg} \cup D^{sat}, \{q\}, \Delta \cup \Delta_{arg} \cup \Delta^{sat*}_d)$. Then, $L(\mathcal{B}') = R^*_\epsilon(L(\mathcal{A}|_p))$ i.e the same as $R^{*\triangleleft}_\epsilon(L(\mathcal{A}|_p))$ except that the rewrite steps are not necessarily leftmost (see [2] for more details and explanations, here only sat differs).

We create another rhs's encoding. So, it permits us to have descendants of instances of rhs's obtained by a leftmost strategy. For example, consider the rhs $c(f(x), g(y))$. We check that instances of $f(x)$ are reduced to their normal forms, by any strategy (thanks to Corollary 2.24), before reducing instances of $g(y)$ by a leftmost strategy.

Definition 3.6 For this, we consider the set of states:

$$D_{spec} = D \cup D^{sat} \cup D^{sat} \times Q_{irr}$$

and the following set of transitions where $(d^{\cdots}_{\cdots} q^{irr}$ denotes the pair $(d^{\cdots}_{\cdots}, q^{irr}))$:

$$\begin{aligned}
\Delta_{spec} = {} & \textstyle\bigcup_{l_i \to r_i \in R} \bigcup_{p \in Pos(r_i)/(PosF(r_i) \cup Arg(r_i))} \bigcup_{k \in \{1,\ldots,ar(r_i(p))\}} \\
& \{r_i(p)(X_1, \ldots, X_n) \to d^{i,p}_{sat,\sigma_1 \cup \ldots \cup \sigma_n} \mid \forall j,\ \sigma_j \text{ is any } Q'\text{-substitution} \\
& \qquad \text{s.t. } dom(\sigma_j) = Var(r_i|_{p.j}), \\
& \qquad \text{where } \forall j,\ X_j = \left| \begin{array}{l} \left. \begin{array}{l} x\sigma_j, q^{irr} \text{ if } j < k \\ x\sigma_j \text{ otherwise} \end{array} \right| \ r_i(p.j) \text{ is any variable } x \\ d^{i,p.j}_{sat,\sigma_j} \text{ if } j = k \\ d^{i,p.j}_{sat,\sigma_j} q^{irr} \text{ if } j < k \\ d^{i,p.j}_{\sigma_j} \text{ otherwise} \end{array} \right. \quad \} \\
& \cup \{r_i(p)(X_1, \ldots, X_n) \to d^{i,p}_{sat,\sigma} \mid l_i \to r_i \in R,\ p \in PosF(r_i), \\
& \qquad \sigma \text{ is any } Q'\text{-substitution s.t. } dom(\sigma) = Var(r_i|_p) \\
& \qquad \text{where } \forall j,\ X_j = \left| \begin{array}{l} x\sigma \mid r_i(p.j) \text{ is any variable } x \\ q^{i,p.j} \in Q_{arg} \text{ otherwise} \end{array} \right. \quad \} \\
& \cup \{x\sigma \to d^{i,\epsilon}_{sat,\sigma} \mid l_i \to r_i \in R,\ r_i \text{ is any variable } x, \\
& \qquad \sigma \text{ is any } Q'\text{-substitution s.t. } dom(\sigma) = \{x\}\}
\end{aligned}$$

Thus, $r_i\sigma$ is also recognized into the state $d^{i,\epsilon}_{sat,\sigma}$.

Now, we define an automaton that recognizes $R^{*\triangleleft}_p(L(\mathcal{A}))$.

Definition 3.7 We define $\mathcal{B}^{\triangleleft}$ an automaton s.t.:

$$\mathcal{B}^{\triangleleft} = (C \cup F, Q^{\triangleleft}, Q_f^{\triangleleft}, \Delta^{\triangleleft})$$

where $Q^{\triangleleft} = Q \cup D_{spec} \cup Q_{arg} \cup (Q \cup Q_{arg}) \times Q_{irr}$ and $Q_f = \{q\}$ and $\Delta^{\triangleleft} = \Delta_d \cup \Delta_d^{sat*} \otimes \Delta_{irr} \cup \Delta_{spec} \cup \Delta^{sat} \cup \Delta_{arg}$.
$\Delta_d^{sat*} \otimes \Delta_{irr}$ is obtained by running the automaton intersection algorithm on transition sets Δ_d^{sat*} and Δ_{irr}. Thus it encodes normal-forms of instances of rhs's.

Lemma 3.8 $L(\mathcal{B}^{\triangleleft}) = R_{\epsilon}^{*\triangleleft}(L(\mathcal{A}|_p))$.

Proof. It comes from Corollary 2.24 and [2]. $\square$

Corollary 3.9 $L(\mathcal{A}[\mathcal{B}^{\triangleleft}]_p) = R_p^{*\triangleleft}(L(\mathcal{A}))$.

Theorem 3.10 *Let R be a rewrite system satisfying restrictions 1,2,3,and 5, and E be the set of data-instances of a given linear term t. If E is recognized by an automaton that discriminates position p into the state q, and possibly p' into q' for some $p' \in \overline{Pos}(t)$ s.t. $p' \not\geq p$, and some states q', then so is $R_p^{*\triangleleft}(E)$.*

Example 3.11 Let $R = \{f(x) \xrightarrow{r1} s(x),\ h(x,y) \xrightarrow{r2} c(f(x), g(y)),\ g(x) \xrightarrow{r3} s(x)\}$ and $t = h(x,y)$. We consider only instances of t by constructors a, s, i.e. $E = \{h(s^*(a), s^*(a))\}$.
We are looking for an automaton that recognized $R_{\epsilon}^{*\triangleleft}(E)$. For sake of simplicity we denote by σ any substitution.
The only leftmost derivation is: $E \to_{[\epsilon,r2,\sigma]} c(f(s^*(a)), g(s^*(a))) \to_{[1,r1,\sigma]} c(s(s^*(a)), g(s^*(a))) \to_{[2,r3,\sigma]} c(s(s^*(a)), s(s^*(a)))$. In the following, we give only sets of transitions.

$\Delta_{t\theta} = \{a \to q_{data},\ s(q_{data}) \to q_{data},\ h(q_{data}, q_{data}) \to q^{\epsilon}\}$ where final state is q^{ϵ}. Let us define $\mathcal{B}^{\triangleleft}$.
$\Delta_d = \{c(d_{\sigma}^{r2,1}, d_{\sigma}^{r2,2}) \to d_{\sigma}^{r2,\epsilon},\ f(q_{data}) \to d_{\sigma}^{r2,1},\ g(q_{data}) \to d_{\sigma}^{r2,2},\ s(q_{data}) \to d_{\sigma}^{r1,\epsilon},\ s(q_{data}) \to d_{\sigma}^{r3,\epsilon}\}$.
$\Delta_{spec} = \{c(d_{sat,\sigma}^{r2,1}, d_{\sigma}^{r2,2}) \to d_{sat,\sigma}^{r2,\epsilon}, c(d_{sat,\sigma}^{r2,1} q^{irr}, d_{sat,\sigma}^{r2,2}) \to d_{sat,\sigma}^{r2,\epsilon},\ f(q_{data}) \to d_{sat,\sigma}^{r2,1},\ g(q_{data}) \to d_{sat,\sigma}^{r2,2},\ s(q_{data}) \to d_{sat,\sigma}^{r1,\epsilon},\ s(q_{data}) \to d_{sat,\sigma}^{r3,\epsilon}\}$.
$\Delta_d^{sat} = \{c(d_{sat,\sigma}^{r2,1}, d_{sat,\sigma}^{r2,2}) \to d_{sat,\sigma}^{r2,\epsilon},\ f(q_{data}) \to d_{sat,\sigma}^{r2,1},\ g(q_{data}) \to d_{sat,\sigma}^{r2,2},\ s(q_{data}) \to d_{sat,\sigma}^{r1,\epsilon},\ s(q_{data}) \to d_{sat,\sigma}^{r3,\epsilon}\}$.
$\Delta^{sat} = \{d_{sat,\sigma}^{r2,\epsilon} \to q^{\epsilon},\ d_{sat,\sigma}^{r1,\epsilon} \to d_{sat,\sigma}^{r2,1},\ d_{sat,\sigma}^{r3,\epsilon} \to d_{sat,\sigma}^{r2,2}\}$.
For the following set, we give only what we will use: $\Delta_d^{sat*} \otimes \Delta_{irr} \supseteq \{s(q_{data}) \to d_{sat,\sigma}^{r3,\epsilon} q^{irr}$,
$s(q_{data}) \to d_{sat,\sigma}^{r1,\epsilon} q^{irr},\ d_{sat,\sigma}^{r1,\epsilon} q^{irr} \to d_{sat,\sigma}^{r2,1} q^{irr}\}$.

Leftmost descendants are recognized indeed. In particular, the non-leftmost descendants $c(f(s^*(a)), s(s^*(a)))$ are not recognized because $s(s^*(a))$ is recognized into the state $d_{sat,\sigma}^{r2,2}$. This state appears in a lhs only in transition $c(d_{sat,\sigma}^{r2,1} q^{irr}, d_{sat,\sigma}^{r2,2}) \to d_{sat,\sigma}^{r2,\epsilon}$ of Δ_{spec} and in a transition of Δ_d^{sat} (but transitions of Δ_d^{sat} do not belong to the final set of transitions, see previous definition). And using $c(d_{sat,\sigma}^{r2,1} q^{irr}, d_{sat,\sigma}^{r2,2}) \to d_{sat,\sigma}^{r2,\epsilon}$ requires that the first argument is normalized, which does not hold for $f(s^*(a))$.

4 An automaton that recognizes $R^*_{left}(E)$

We look for an automaton that recognizes the sets of descendants according to a leftmost strategy.

Recall that R is assumed to satisfy Restriction 5' (left-variable preserving). We can transform it so that Restriction 5 (variable preserving) is satisfied, in the following way. A new binary constructor eat is introduced, and $eat(t, t')$ intuitively means that we want to keep t as the result, and to get rid of t'. Because of the leftmost strategy, the term to be kept has to be the left argument of eat. By introducing eat into the rhs, we can transform a non-variable preserving rule into a variable-preserving one. Then, by introducing more rewrite rules, we extend the existing defined-functions to take the new constructor eat into account. The method is explained by the following example, and the algorithms are given in the appendix B.1.

Example 4.1 Let $R = \{f(s(s(x)), y) \to x\}$.
After running algorithm, we obtain the following TRS :
$R = \{f(s(s(x)), y) \to eat(x, y),\ f(eat(s(s(x)), x_1), y) \to eat(x, eat(x_1, y)),$ $f(s(eat(s(x), x_{1.1})), y) \to eat(x, eat(x_{1.1}, y)), f(eat(s(eat(s(x), x_{1.1.1})), x_1), y) \to$ $eat(x, eat(x_{1.1.1}, eat(x_1, y)))\}$.

Definition 4.2

$$R^*_{left,p}(L) = \{s' \mid \exists s \in L,\ s \to^*_{[u_1,\ldots,u_n]} s' \text{ by a leftmost strat. with } \forall i,\ u_i \geq p\}.$$

Lemma 4.3 *Let R be a rewrite system satisfying restrictions 1 to 5, and E be the set of data-instances of a given linear term t.*
Let $p \in\ PosF(t)$, and L be a language s.t. $L|_p = E|_p$, and that is recognized by an automaton that discriminates every position $p' \in\ PosF(t) \mid p' \geq p$. Then,

$$R^*_{left,p}(L) = R^{*\lhd}_p(L) \quad if\ Succ(p) = \emptyset$$

Otherwise, let $Succ(p) = \{p_1, \ldots, p_n\}$ with $p_1 \lhd \ldots \lhd p_n$

$$R^*_{left,p}(L) = \left| \begin{array}{l} R^{*\lhd}_p[R^*_{left,p_1}(L)] \\ \cup\ R^{*\lhd}_p[R^*_{left,p_2}((R^*_{left,p_1}(L) \cap IRR_{p_1}(R)))] \\ \cup \ldots \cup\ \ R^{*\lhd}_p[R^*_{left,p_n}(\ldots(R^*_{left,p_1}(L) \cap IRR_{p_1}(R)) \ldots \cap IRR_{p_{n-1}}(R))] \end{array} \right.$$

and $R^*_{left,p}(L)$ is recognized by an automaton $\mathcal{A}'$ s.t. if $p' \in\ \overline{Pos}(t)$, $p' \not\geq p$ and $\mathcal{A}$ discriminates p' into q', then $\mathcal{A}'$ also discriminates p' into q'.

Proof. See appendix B. □

Theorem 4.4 *Let R be a rewrite system satisfying restrictions 1 to 5, and E be the set of data-instances of a given linear term t.*

$$R^*_{left}(E) = R^*_{left,\epsilon}(E) \quad if\ t(\epsilon) \in F$$

Otherwise, let $Succ(p) = \{p_1, \ldots, p_n\}$ with $p_1 \triangleleft \ldots \triangleleft p_n$

$$R^*_{left}(E) = \left| \begin{array}{l} R^*_{left,p_1}(E) \\ \cup\ R^*_{left,p_2}((R^*_{left,p_1}(L) \cap IRR_{p_1}(R))) \\ \cup \ldots \cup R^*_{left,p_n}(\ldots(R^*_{left,p_1}(L) \cap IRR_{p_1}(R)) \ldots \cap IRR_{p_{n-1}}(R)) \end{array} \right.$$

and $R^*_{left}(E)$ is effectively recognized by an automaton.

Proof. See appendix B. □

To remove eat, we replace each transition of the form $eat(q, q') \to q''$ by $q \to q''$.

Example 4.5 Let $R = \{f(x) \to s(x),\ h(x, y) \to c(f(x), g(y)),\ g(x) \to s(x)\}$ and $E = \{h(g(s^*(a)), f(s^*(a)))\}$. For clarity, we denote $s^*(a)$ by $*$.
$R^*_{left}(E) = R^*_{left,\epsilon}(E) = R^{*\triangleleft}_{\epsilon}(R^*_{left,1}(E)) \cup R^{*\triangleleft}_{\epsilon}(R^*_{left,2}(R^*_{left,1}(E) \cap IRR_1(R))$.

- $R^*_{left,1}(E) = R^{*\triangleleft}_1(E)$
  $= E \cup \{h(s(*), f(*))\}$ denoted by L_1.
- $R^*_{left,2}(R^*_{left,1}(E) \cap IRR_1(R)) = R^*_{left,2}(\{h(s(*), f(*))\}$
  $= \{h(s(*), f(*)\} \cup \{h(s(*), s(*)\}$ denoted by L_2

Finally, we obtain leftmost descendants which are:
$L_1 \cup L_2 \cup \{c(f(g(*)), g(f(*)))\} \cup \{c(s(g(*)), g(f(*)))\} \cup \{c(s(s(*)), g(f(*)))\} \cup \{c(s(s(*)), s(f(*)))\} \cup \{c(s(s(*)), s(s(*)))\} \cup \{c(f(s(*)), g(s(*)))\} \cup \{c(s(s(*)), g(s(*)))\} \cup \{c(f(s(*)), g(f(*)))\}$.

5 Conclusion

When computing descendants, taking the leftmost strategy into account is possible, keeping the same class of tree languages (the regular ones) and assuming the same restrictions as for innermost strategy [3], plus others (restrictions iv, v'). Restriction iv (no permutative rules) cannot be weakened easily, because our method computes left-basic derivations, instead of computing leftmost derivations directly. However, we think that without this restriction, the set of leftmost descendants is still regular.

References

[1] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata techniques and Applications* (TATA). http://l3ux02.univ-lille3.fr/tata.

[2] P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proceedings of the 6th international conference on Logic for Programming and Automated Reasoning (LPAR), Tbilisi (Republic of Georgia)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1999.

[3] P. Réty and J. Vuotto. Regular sets of descendants by some rewrite strategies. In *Proceedings of 13th Conference on Rewriting Techniques and Applications, Copenhagen (Denmark),* LNCS. Springer-Verlag, 2002.

[4] P. Réty. Méthodes d'unification par surreduction. Thèse de doctorat de l'université de Nancy 1.

Appendix

A Proofs on Rewriting Commutation and Left-basic Derivation

A.1 Proof of Lemma 2.15

The fact that p has no antecedent it's obvious because of Definition 2.8. Either p occurs on the right of p_0, either p occurs on top of p_0.
Now, let us show that (1) $leftmot \Rightarrow$ (2) $leftmost$.
Let (1) leftmost. Let us classify $Var(r)$ from left to right. if $|Var(r)| = n$ then $Var(r) = \{x_1, \ldots, x_n\}$. Since (1) is leftmost, let us denote by $x_i = x$ the first variable instantiated by a term containing a defined-function. Then, $\forall j \in [1..i[,\ \sigma(x_j)$ do not contain function.
Let us suppose that p_0 are not a leftmost position in s. This is possible only if $\exists$ a function that occurs on the left of x in l instantiated by a term containing a function. Now it happens that it is not possible because of prohibition of permutative rules (see restriction iv) and because of variable preserving TRS (see restriction v). Then, if we classify $Var(l)$ from left to right, we obtain the same order as $Var(r)$. Hence, (2) is leftmost.

A.2 Proof of Lemma 2.16

This proof is based on Lemma 2.15.
Let $ant(p_n, t_{n-1}) = q_0 \ \Rightarrow t_{n-1} \rightarrow_{[p_{n-1}, l_{n-1} \rightarrow r_{n-1}]} t_n \rightarrow_{[p_n, l_n \rightarrow r_n]} t_{n+1}$ commute itself into $t_{n-1} \rightarrow_{[q_0, l_n \rightarrow r_n]} t'_n \rightarrow_{[p_{n-1}, l_{n-1} \rightarrow r_{n-1}]} t_{n+1}$. And p_{n-1} has not antecedent in t_{n-1}, and it is leftmost.
Let $ant(q_0, t_{n-2}) = q_1 \ \Rightarrow t_{n-2} \rightarrow_{[p_{n-2}, l_{n-2} \rightarrow r_{n-2}]} t_{n-1} \rightarrow_{[q_0, l_n \rightarrow r_n]} t'_n$ commute itself into $t_{n-2} \rightarrow_{[q_1, l_n \rightarrow r_n]} t'_n \rightarrow_{[p_{n-2}, l_{n-2} \rightarrow r_{n-2}]} t'_n$. And p_{n-2} has not antecedent

in t_{n-2}, and it is leftmost.
...
By induction, let $ant(q_{n-2}, t_0) = q_{n-1} \Rightarrow t_0 \rightarrow_{[p_0, l_0 \rightarrow r_0]} t_1 \rightarrow_{[q_{n-2}, l_n \rightarrow r_n]} t'_2$ commute itself into $t_0 \rightarrow_{[q_{n-1}, l_n \rightarrow r_n]} t'_1 \rightarrow_{[p_0, l_0 \rightarrow r_0]} t'_2$. And p_0 has not antecedant in t_0 and it is leftmost.
Hence, we obtain (2).

A.3 Proof of Lemma 2.19

Let (1) a left-basic derivation.
Let us suppose that $\exists i \in \{1 \ldots n\}$ s.t. p_i has an antecedent in t_{i-1}. By Definition 2.17, $p_i \in B(t_i)$ where $\forall j,\ B(t_i) = \{p' \mid p' \leq p_{i-1}\} \cup \{p_{i-1}.v \mid v \in PosF(r_{i-1})\} \cup \{p' \mid p_{i-1} \lhd p'\}$. Let $ant(p_i, t_{i-1}) = q$.
By Definition 2.8,

(i) $q \in Pos(t_{i-1})$

(ii) - $p_i \lhd p_{i-1}$ *and* $p_i = q$ or, (A)
- $\exists p' \in PosVar(r_{i-1})$ *with* $r_{i-1}|_{p'} = x$ *s.t.* $p_i = p_{i-1}.p'.w$ *and* $q = p_{i-1}.p''.w$ *where* p'' is an occurrence of variable x in l_{i-1}. (B)

It happens that (B) is impossible because of Definition 2.17; p_i would occur in forbidden position. (A) is impossible too, $not(p_i \lhd p_{i-1})$ else the derivation is not left-basic.
Hence, $\forall i \in \{1 \ldots n\},\ p_i$ has not an antecedent in t_{i-1}.

Let $\forall i \in \{1 \ldots n\},\ p_i$ has not an antecedent in t_{i-1}.
$not(p_i \lhd p_{i-1})$ else we find an antecedent, and $p_i \neq p_{i-1}.p'.w$ where $p' \in PosVar(r_{i-1})$
$(r_{i-1}|_{p'} = x)$ because else we find $q = p_{i-1}.p''.w$ where p'' is an occurrence of x in p_{i-1}. Hence, it is left-basic.

A.4 Proof of Lemma 2.20

Let $ant(p_n, t_{n-1}) = q_0$,. Derivation can be commuted and we obtain:
$t_0 \rightarrow t_1 \rightarrow \ldots t_j \rightarrow t_{n-1} \rightarrow_{[q_0]} t'_n \rightarrow_{[p_{n-1}]} t_{n+1}$ where p_{n-1} has not antecedent in t_{n-1} according to Lemma 2.15.
By induction, let $j < n$ the smaller integer s.t. p_n admits an antecedent in t_j. Let denote by q this antecedent. By Lemma 2.16, we can commute and we obtain:
$t_0 \rightarrow \ldots t_j \rightarrow_{[q, l_n \rightarrow r_n]} t'_{j+1} \rightarrow_{[p_j]} \ldots \rightarrow t'_n \rightarrow_{[p_{n-1}]} t_{n+1}$ (4) and p_j has not antecedent in t_j.
q has not an antecedent in t_{j-1} since according to the remark $j < n$ is the smaller integer s.t. p_n admits an antecedent in t_j. Moreover, since (1) is left-basic, $t_0 \rightarrow \ldots t_j$ is left-basic. And since $\forall i \in \{j+1 \ldots n-1\}$ p_i has not antecedent in t_i for the derivation $t_j \rightarrow_{[q]} t'_{j+1} \rightarrow_{[p_j]} \ldots \rightarrow t'_n \rightarrow_{[p_{n-1}]} t_{n+1}$ (3) then (3) is left-basic according to Lemma 2.19. And, since q has not antecedent in t_{j-1} then (4) is left-basic.

A.5 Proof of Lemma 2.21

Let $i \in \{1 \dots n\}$ s.t. $t_0 \rightarrow \dots t_i$ left-basic derivation and $t_0 \rightarrow \dots t_i \rightarrow t_{i+1}$ (1') non-left-basic.
By running of Lemma 2.20, (1') can be commuted in $t_0 \rightarrow \dots t_j \rightarrow_{[q]} t'_{j+1} \rightarrow \dots t'_i \rightarrow t_{i+1}$ (2') with $ant(p_i, t_j) = q$ and (2') is left-basic. And by Lemma 2.16, (2') is leftmost if (1') is leftmost.
We use Lemma 2.20, many times are necessary, and so $t_0 \rightarrow^* t_n$ by a left-basic derivation. We proceed in the same way with Lemma 2.16, and (2) is leftmost if (1) is leftmost.

A.6 Proof of Lemma 2.22

Let $t_0 \rightarrow_{[p_0, l_0 \rightarrow r_0]} t_1 \rightarrow \dots t_n \rightarrow_{[p_n, l_n \rightarrow r_n]} t_{n+1}$ (1) a left-basic derivation, and let $j \in \{0, \dots, n\}$.
Let $p' \in \overline{Pos}(t_j) - B(t_j)$ s.t. $t_j(p') \in \mathcal{F}$. Let us take $k = j + 1$. By absurd, let us suppose that $\neg(\exists p'_k \in \overline{Pos}(t_k) - B(t_k)$ s.t. $t_k|_{p'_k} = t_j|_{p'})$. I.e. $\forall p'_k \in \overline{Pos}(t_k) - B(t_k)$ s.t. $t_k|_{p'_k} \neq t_j|_{p'}$. Then, we have to remove $t_j|_{p'}$ during the rewriting step.

- if $p' \lhd p_j$ then this is impossible.
- if p' below p_j, $p' \in PosVar(l_j)$, seeing the constructor-based system, then the rewrite rule $l_j \rightarrow r_j$ should have to eliminate variables. It happens that we have a restrictions removing this model of rules.

By induction, we obtain the result for $k = j + 2, \dots, n$.

A.7 Proof of Theorem 2.23

By absurd, let us suppose that $\exists j, p'$ s.t. $t_j \rightarrow_{[p']} u$ with $p' \lhd p_j$. According to Lemma 2.22, $\not\exists k > j$ s.t. $p_k = p'$. Then, $p' \notin B(t_{j+1})$ and $t_{j+1} \in \mathcal{F}$. According to Lemma 2.22, $\exists p'_{n+1} \in Pos(t_{n+1})$ s.t. $t_{n+1}|_{p'_{n+1}} = t_j|p'$. That contradicts the fact as t_{n+1} be normalized.

B Proofs on Computation of Leftmost Descendants

B.1 Transforming R to satisfy restriction v

$construct\_r(l \rightarrow r)$
  $EraseVar := Var(l) - Var(r)$ *where* $|Var(l)| = n$ *and* $|Var(r)| = m$.
  $r := eat(r, x_{m+1})$
  For $i := m + 2$ *to* n *do* $r := r[2 \leftarrow eat(r|_2, x_i)]$

In the following function, $shift(stack, p)$ raise all positions which are under p in stack by a depth of 1. For example, if $p = 2$ and, 2.1, 2.2 and 1 are in stack, after running this function, 2.1.1, 2.1.2 and 1 are in stack.

$shift(stack, p)$

$\quad \forall q \in stack \; s.t. \; \exists w \; and \; q = p.w \; replace \; q \; by \; p.1.w$

$build\_l(l, stack, R')$

$\quad p := head(stack); \quad stack := pop(stack);$

$\quad if \; not \; empty(stack) \; then \; build(l, stack, R')$

$\quad l' := l[p \leftarrow eat(l|_p, x_p)]; \quad r' := construct\_r(l' \rightarrow r); \quad Add \; l' \rightarrow r' \; to \; R'$

$\quad stack := shift(stack, p); \quad if \; not \; empty(stack) \; then \; build(l', stack, R')$

$Transfom(R)$

$\quad init : R' = \emptyset$

$\quad \forall l \rightarrow r \in R, \; if \; Var(l) = Var(r) \; then \; add \; l \rightarrow r \; to \; R'$

$\qquad ifVar(l) \neq Var(r) \; then \; r' := construct\_r(l \rightarrow r);$

$\qquad\qquad add \; l \rightarrow r' \; to \; R'$

$\qquad$ let $C(l)$ stack that contains all constructors of l.

$\qquad if|C(l)| \neq \emptyset \; then \; build(l, C(l), R')$

$\quad R := R'$

B.2 Proof of Lemma 4.3

- If $Succ(p) = \emptyset$, then $\forall s \in L, \forall p' \in Pos(s), (p' > p \implies s(p') \in C)$. p is the only one function position of $s|_p$. In rhs's, it may have several function position that are incomparable (in opposition with nested). $R_p^{*\triangleleft}(L)$ compute leftmostly. Therefore, $R_p^{*\triangleleft}(L) = R^*_{left,p}(L)$.
  We get $\mathcal{A}'$ by Theorem 3.10.
- Let $Succ(p) = \{p_1, \ldots, p_n\}$ with $p_1 \triangleleft \ldots \triangleleft p_n$. Let $s \in L$. $\forall \, i \in \{1, \ldots, n\}$, s can be rewritten leftmostly at position p_i, but descendants at position p_{i-1} must have computed and normalized at this position before, since $p_{i-1} \triangleleft p_i$. Let us note L_1 set of terms obtained from s after leftmost rewriting at position p_1,. . ., L_n set of terms obtained from s after leftmost rewriting at position p_n. We have:
  $L_1 = R^*_{left,p_1}(L)$
  $L_2 = R^*_{left,p_2}(R^*_{left,p_1}(L) \cap IRR_{p_1}(R))$
  ...
  $L_n = R^*_{left,p_n}(L_{n-1} \cap IRR_{p_{n-1}}(R))$.
  Remark : Those descendants are obtained by left-basic rewriting.
  $L_1, \ldots, L_n$ can be possibly rewritten at position p. $R_p^{*\triangleleft}(L_1) \cup \ldots \cup R_p^{*\triangleleft}(L_n) = R^*_{left,p}(L)$.
  L is recognized by an automaton $\mathcal{A}$ that discriminates every position $p' \in PosF(t)$ s.t. $p' \geq p$ and so, since $\forall i \in \{1, \ldots, n\}$, $p_i > p$, every position p' s.t. $p' \geq p_i$.
  By induction hypothesis, L_1 is recognized by an automaton $\mathcal{A}_1$ that discriminates p and every position $p' \in PosF(t)$s.t. $p' \not> p_1$ and so, in particular, every position $p' \geq p_i$ $\forall i \in \{2, \ldots, n\}$ and every position $p' \not> p$. By Theorem 3.10, $R_p^{*\triangleleft}(L_1)$ is recognized by an automaton that

discriminates positions $p' \not> p$.
By Theorem 2.14, $IRR_{p_i}(R)$ is recognized by an automaton that discriminates every position $p' \in PosF(t)$ s.t. $p' \not> p_i$ ($p' \not> p_1$ for $IRR_{p_1}(R), \ldots,$ $p' \not> p_n$ for $IRR_{p_n}(R)$). For $j \in \{1, \ldots, n\}$, let us suppose that L_{j-1} is recognized by an automaton that discriminates every position $p' \not> p_{j-1}$ (i.e. positions that are discriminated before computing $R^*_{left,p_{j-1}}(\ldots)$ minus positions that are below p_{j-1}). By Lemma 2.5, $L_{j-1} \cap IRR_{p_{j-1}}(R)$ is recognized by an automaton that discriminates every position $p' \not> p_{j-1}$, so in particular $p' \geq p_j$. L_j is recognized by an automaton that discriminates every position $p' \not> p_j$ (i.e. positions that are discriminated before computing $R^*_{left,p_j}(\ldots)$ minus positions that are below p_j) and in particular every position $p' \not> p$. By Lemma 3.10, $R^{*\lhd}_p(L_j)$ is recognized by an automaton that discriminates every position $p' \not> p$.
Finally, by union, we obtain an automaton that discriminates p and preserves discrimination of positions $p' \not> p$.

B.3 Proof of Theorem 4.4

We have two cases:

- If $\epsilon \in PosF(t)$, obviously $R^*_{left}(E) = R^*_{left,\epsilon}(E)$.
- If $\epsilon \notin PosF(t)$, and $Succ(p) = \{p_1, \ldots, p_n\}$ *with* $p_1 \lhd \ldots \lhd p_n$, $\forall\ i, j$ s.t. $p_i \lhd p_j$, leftmost descendants at position p_j can be computed after have normalized those at position p_i. Then obviously,$R^*_{left}(E) =$ $R^*left, p_1(E) \cup \ldots$
  $\cup\ R^*left, p_{n-1}(\ldots(R^*left, p_1(L) \cap IRR_1(R)))\ldots) \cup \ldots$
  $\cup\ R^*left, p_n(\ldots(R^*left, p_1(L) \cap IRR_1(R)))\ldots)$

Term Rewriting with Type-safe Traversal Functions

M.G.J. van den Brand P. Klint J.J.Vinju

Centrum voor Wiskunde en Informatica
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands

Abstract

Term rewriting is an appealing technique for performing program analysis and program transformation. Tree (term) traversal is frequently used but is not supported by standard term rewriting. In this paper, many-sorted first-order term rewriting is extended with automatic tree traversal by adding two primitive tree traversal strategies and complementing them with three types of traversals. These so-called traversal functions can be either top-down or bottom-up. They can be sort preserving, mapping to a single sort, or a combination of these two. Traversal functions have a simple design, their application is type-safe in a first-order many-sorted setting and can be implemented efficiently. We describe the operational semantics of traversal functions and discuss applications.

1 Introduction

Program analysis and program transformation usually take the syntax tree of a program as starting point. Operations on this tree can be expressed in many ways, ranging from imperative or object-oriented programs, to attribute grammars and rewrite systems. One common problem that one encounters is how to express the *traversal* of a tree: to visit all the nodes of a tree and extract information from some nodes or make changes to certain other nodes.

The kinds of nodes that may appear in a program's syntax tree are determined by the grammar of the language the program is written in. Typically, each rule in the grammar corresponds to a node category in the syntax tree. Real-life languages are described by grammars containing a few hundred up to one thousand grammar rules. This immediately reveals a hurdle for writing tree traversals: a naive recursive traversal function should consider many node categories and the size of its definition will grow accordingly. This becomes even more dramatic if we realize that the traversal function will only do some real work (apart from traversing) for very few node categories.

This problem asks for a form of automation that takes care of the tree traversal itself so that the human programmer can concentrate on the few node categories where real work is to be done. Stated differently, we want to offer tree traversal as a built-in, not a burden, to the programmer.

From previous experience [3,5,6] we know that term rewriting is a convenient, scalable technology for expressing analysis, transformation, and renovation of individual programs and complete software systems. In this paper we address therefore the question how tree traversals can be added to the term rewriting paradigm.

One important requirement is to have a typed design of automated tree traversals, such that terms are always well-formed. Another requirement is to have simplicity of design and use. In particular, we prefer to remain inside the domain of first-order specifications.

In the remainder of this introduction we will briefly recapitulate term rewriting, and discuss why traversal functions are necessary in term rewriting. In Section 2 we extend the algebraic specification formalism ASF+SDF [1,9] with traversal functions and in Section 3 we give some small examples of the use of these traversal functions. The operational semantics of traversal functions is given in Section 4. Section 5 describes the experience with traversal functions and Section 6 gives a discussion and also discusses related work.

A brief recapitulation of term rewriting

Algorithm 1 An interpreter for innermost rewriting.

```
funct innermost(term, rules) ≡
  (fn, children) := decompose(term)
  children' := nil
  foreach child in children do
    children' := append(children', innermost(child, rules))
  od
  term := compose(fn, children')
  reduct := reduce(term, rules)
  return if reduct = fail then term else reduct fi.
funct reduce(term, rules) ≡
  foreach rule in rules do
    (lhs, rhs) := decompose(rule)
    bindings := match(term, lhs)
    if bindings ≠ fail then
      return innermost(substitute(rhs, bindings), rules)
    fi
  od
  return fail .
```

A basic insight in term rewriting is important for understanding the traversal functions described in this paper. Therefore we give a brief and informal recapitulation of innermost term rewriting. For a full account see [11].

A *term* is a prefix expression consisting of constants (e.g., `a` or `12`), variables (e.g., `X`) or function applications (e.g., `f(a, X, 12)`). For simplicity, we will view constants as nullary functions. A *closed term* is a term without variables. A *rewrite rule* is a pair of terms $T_1 \rightarrow T_2$. Both T_1 and T_2 may contain variables provided that each variable in T_2 also occurs in T_1. A term *matches*

```
module Tree-syntax
imports Naturals
exports
  sorts TREE
  context-free syntax
    NAT             -> TREE
    f(TREE, TREE) -> TREE
    g(TREE, TREE) -> TREE
    h(TREE, TREE) -> TREE
```

Figure 1: SDF grammar for simple tree language.

another term if it is structurally equal modulo occurrences of variables[1] (e.g., `f(a, X)` matches `f(a, b)`) and results in a *binding* (e.g., `X` is bound to `b`). The bindings resulting from matching can be used for *substitution*, i.e., replace the variables in a term by the values they are bound to.

Given a closed term T and a set of rewrite rules, the purpose of a rewrite rule interpreter is to find a sub-term that can be reduced: the so-called *redex*. If sub-term R of T matches with the left-hand side of a rule $T_1 \to T_2$, the bindings resulting from this match can be substituted in T_2 yielding T_2'. R is then replaced in T by T_2' and the search for a new redex is continued. Rewriting stops when no new redex can be found and we say that the term is then in *normal form*.

Different methods for selecting the redex may yield different results. In this paper we limit our attention to leftmost innermost rewriting in which the redex is searched in a left-to-right, bottom-up fashion.

The operation of a rewrite rule interpreter is shown in more detail in Algorithm 1. The functions *match* and *substitute* are not further defined, but have a meaning as just sketched. The functions *decompose* and *compose* manipulate terms and also rules, and *append* appends an element to the end of a list.

Note that the underlying term representation can be either typed or untyped. The *match*, *substitute*, *decompose* and *compose* functions have to take this into account.

Observe how function *innermost* first reduces the children of the current term before attempting to reduce the term itself. This realizes a bottom-up traversal of the term. Also note that if the reduction of the term fails, it returns itself as result. The function *reduce* performs, if possible, one reduction step. It searches all rules for a matching left-hand side and, if found, the bindings resulting from the successful match are substituted in the corresponding right-hand side. This modified right-hand side is then further reduced with innermost rewriting.

In Section 4 we will extend Algorithm 1 to cover traversal functions as well.

[1] A matching algorithm has to do some extra work if a specific variable name occurs more than once in a pattern. Usually the different occurrences are renamed to fresh variables and equality checks are added.

Why traversal functions in term rewriting? Rewrite rules are very convenient to express transformations on trees and one may wonder why traversal functions are needed at all. We will clarify this by way of simple trees containing natural numbers. Figure 1 displays an Sdf grammar [10] for a simple tree language. The leafs are natural numbers and the nodes are constructed with one of the binary constructors `f`, `g` or `h`. Transformations on these trees can now be defined easily. For instance, if we want to replace all occurrences of `f` by `h`, then the following rule suffices:

```
[t1] f(T1, T2) = h(T1, T2)
```

Applying this rule to the term `f(f(g(1,2),3),4)` leads to a normal form in two steps (using innermost reduction):

```
f(f(g(1,2),3),4) -> f(h(g(1,2),3),4) -> h(h(g(1,2),3),4)
```

Similarly, if we want to replace all sub-trees of the form `f(g(T1, T2), T3)` by `h(T1, h(T2, T3))`, we can achieve this by the single rule:

```
[t2] f(g(T1, T2), T3) = h(T1, h(T2, T3))
```

If we apply this rule to `f(f(g(1,2),3),4)` we get a normal form in one step:

```
f(f(g(1,2),3),4) -> f(h(1,h(2,3)),4)
```

Note how in both cases the standard (innermost) reduction order of the rewriting system takes care of the complete traversal of the term. Such elegant specifications have, however, three severe limitations.

- First, if we want to have the combined effect of rules `[t1]` and `[t2]`, we get unpredictable results, since the two rules interfere with each other: the combined rewrite system is said to be *non-confluent*. Applying the above two rules to our sample term `f(f(g(1,2),3),4)` may lead to either `h(h(g(1,2),3),4)` or `h(h(1,h(2,3)),4)` in two steps, depending on whether `[t1]` or `[t2]` is applied in the first reduction step.
- The second problem is that rewrite rules cannot access any context information. In other words a rewrite rule has only one parameter, which is the term that matches the left-hand side. Especially for program transformation this is very limiting.
- Thirdly, in ordinary (typed) term rewriting only type-preserving rewrite rules are allowed, i.e., the type of the left-hand side of a rewrite rule has to be equal to the type of the right-hand side of that rule. Sub-terms can only be replaced by sub-terms of the same type, thus enforcing that the complete term remains well-typed. In this way, one cannot express non-type-preserving traversals such as the (abstract) interpretation or analysis of a term.

A common solution to the above three problems is to introduce new function symbols that eliminate the interference between rules. In our example, if we introduce the functions `trafo1` and `trafo2`, we can explicitly control the outcome of the combined transformation by the order in which we apply

```
module Tree-trafo12
imports Tree-syntax
exports
context-free syntax
  trafo1(TREE)       -> TREE
  trafo2(TREE)       -> TREE
equations
[0] trafo1(N)         = N
[1] trafo1(f(T1, T2)) = h(trafo1(T1), trafo1(T2))
[2] trafo1(g(T1, T2)) = g(trafo1(T1), trafo1(T2))
[3] trafo1(h(T1, T2)) = h(trafo1(T1), trafo1(T2))

[4] trafo2(N)              = N
[5] trafo2(f(g(T1,T2),T3)) = h(trafo2(T1), h(trafo2(T2), trafo2(T3)))
[6] trafo2(g(T1, T2))      = g(trafo2(T1), trafo2(T2))
[7] trafo2(h(T1, T2))      = h(trafo2(T1), trafo2(T2))
```

Figure 2: Definition of `trafo1` and `trafo2`.

`trafo1` and `trafo2` to the initial term. By introducing extra function symbols, we also gain the ability to pass data around using extra parameters of these functions. This adds *data-flow* to a specification. Finally, the function symbols allow to express non-type-preserving transformations by explicitly typing the function to accept one type and yield another.

So by introducing first-order functions, three limitations of rewrite rules are solved. The main down-side of this approach is that we loose the built-in facility of innermost rewriting to traverse the input term without an explicit effort of the programmer. Extra rewrite rules are needed to define the traversal of `trafo1` and `trafo2` over the input term, as shown in Figure 2. Observe that equations `[2]` and `[5]` in the figure correspond to the original equations `[t1]` and `[t2]`, respectively. The other equations are just needed to define the tree traversal. Defining the traversal rules requires explicit knowledge of *all* productions in the grammar (in this case the definitions of `f`, `g` and `h`). In this example, the number of rules per function is directly related to the size of the language shown in Figure 1. For large grammars this is clearly undesirable.

The traversal functions presented in this paper, solve the problem of the extra rules needed for term traversal without loosing the practical abilities of first-order functions to carry data around and having non-sort-preserving transformations.

2 Traversal functions in Asf+Sdf

We want to automate tree traversal in the many-sorted, first-order term rewriting language ASF+SDF [1,9]. ASF+SDF uses context-free syntax for defining the signature of terms. As a result, terms can be written in arbitrary user-defined notation. The context-free syntax is defined in SDF [2]. Terms are used in

[2] Syntax Definition Formalism.

rewrite rules defined in ASF[3]. For the purpose of this paper, the following features of ASF are relevant:

- Many-sorted (typed) terms.
- Unconditional and conditional rules. Conditions come in three flavors: equality between terms, inequality between terms, and so-called assignment conditions that introduce new variables.
- Default rules that are tried only if all other rules fail.
- Terms are normalized by leftmost innermost reduction.

The idea of traversal functions is as follows. The programmer defines functions as usual by providing a signature and defining rewrite rules. The signature of a traversal function has to be defined as well. This is an ordinary declaration but it is explicitly labeled with the attribute `traversal`. We call such a labeled function a traversal function since from the user's perspective it automatically traverses a term: the rewrite rules for term traversal do not have to be specified anymore since they are provided automatically by the `traversal` attribute. The specification writer only has to give rewrite rules for the nodes that the traversal function will actually visit.

The functionality provided by the `traversal` attribute thus defines the *traversal* behavior while rewrite rules provided by the user define the *visit* behavior for nodes. If during innermost rewriting a traversal function appears as outermost function symbol of a redex, then that function will first be used to traverse the redex before further reductions occur.

Conceptually, a traversal function is a shorthand for a possibly large set of rewrite rules. For every traversal function a set of rewrite rules can be calculated that implements both the traversal and the actual rewriting of some sub-terms. This is a nice way of defining the semantics of traversal functions. More details can be found in [7].

The question is what sort of traversals we can provide in our fully typed setting. We allow three *types* of functions for the visiting behavior and two types of traversal *strategies* which we now discuss in order. The merits and limitations of this approach are discussed in Section 6. For extensive examples we refer the reader to [7].

Traversal functions are partitioned into three *types*, defined as follows:

Transformer: a sort-preserving transformation that will traverse its first argument. Possible extra arguments may contain additional data that can be used (but not modified) during the traversal. A transformer is declared as follows:

$f(S_1, \ldots, S_n) \rightarrow S_1$ {`traversal(trafo)`}

Because a transformer always returns the same sort, it is type-safe. A transformer is used to transform a tree.

[3] Algebraic Specification Formalism.

Accumulator: a mapping of all node types to a single type. It will traverse its first argument, while the second argument keeps the accumulated value. An accumulator is declared as follows:

$f(S_1, S_2, \ldots, S_n) \rightarrow S_2$ {`traversal(accu)`}

After each application of an accumulator, the accumulated argument is updated. The next application of the accumulator, possibly somewhere else in the term, will use the *new* value of the accumulated argument. In other words, the accumulator acts as a global, modifiable state during the traversal.

An accumulator function never changes the tree, only its accumulated argument. Furthermore, the type of the second argument has to be equal to the result type. The end-result of an accumulator is the value of the accumulated argument. By these restrictions, an accumulator is also type-safe for every instantiation. An accumulator is meant to be used to extract information from a tree.

Accumulating transformer: a sort preserving transformation that accumulates information while traversing its first argument. The second argument maintains the accumulated value. The return value of an accumulating transformer is a tuple consisting of the transformed first argument and accumulated value. An accumulating transformer is declared as follows:

$f(S_1, S_2, \ldots, S_n) \rightarrow S_1 \# S_2$ {`traversal(accu, trafo)`}

An accumulating transformer is used to simultaneously extract information from a tree and transform it.

Transformers, accumulators, and accumulating transformers may be overloaded in their first argument to obtain visitors for heterogeneous trees. This means that a single traversal function can visit different types of nodes in a single traversal.

The optional extra arguments of traversal functions can carry information down and their defining rewrite rules can extract information from their children by using conditions. So we can express analysis and transformation using non-local information rather easily.

Having these three types of traversals, they must be provided with traversal strategies. Traversal strategies determine the order of traversal and the "depth" of the traversal. We provide a choice among the following two strategies for each type of traversal.

Bottom-up: the function traverses *all* the sub-trees of a node while its rewrite rules are used to visit the nodes in a *bottom-up* fashion. The annotation `bottom-up` selects this behavior. A traversal function without an explicit indication of a visiting strategy also uses the bottom-up strategy.

Top-down: the function traverses the sub-trees of a node in a top-down fashion and stops recurring at the first node where one of its rewrite rules applies and does not go to any sub-trees of that node. The annotation `top-down` selects this behavior.

A transformer with a `bottom-up` strategy resembles standard innermost rewriting; it is sort preserving and bottom-up. It is as if a small rewriting

```
module Tree-inc
imports Tree-syntax
exports
context-free syntax
  inc(TREE) -> TREE {traversal(trafo)}
equations
[1] inc(N) = N + 1
```

in		out	
in	`inc( f( g( f(1,2), 3 ), g( g(4,5), 6 )) )`	out	`f( g( f(2,3), 4 ), g( g(5,6), 7 ))`

Figure 3: The transformer `inc` increments each number in a tree.

```
module Tree-sum
imports Tree-syntax
exports
context-free syntax
   sum(TREE, NAT) -> NAT {traversal(accu)}
equations
[1] sum(N1, N2) = N1 + N2
```

in		out	
in	`sum( f( g( f(1,2), 3 ), g( g(4,5), 6 )), 0)`	out	`21`

Figure 4: The accumulator `sum` computes the sum of all numbers in a tree.

system is defined within the context of a transformer function. The difference is that a transformer function inflicts one reduction on a node, while innermost reduction normalizes a node completely.

The `top-down` strategy is rather powerful because it stops, allowing the user to continue the traversal under certain conditions.

Note that the horizontal ordering (left-to-right or right-to-left), might also be a parameter. In this work we have focused on left-to-right traversals.

3 Examples

After this general description of traversal functions, it is time to illustrate these concepts. We will give a very simple example of each type of traversal function just described. More realistic examples are presented in Section 5.

Increment the numbers in a tree. The specification in Figure 3 shows the transformer `inc`. Its purpose is to increment all numbers that occur in a tree. The results of applying `inc` to a sample tree are also shown in Figure 3.

Add the numbers in a tree. The second example shows the use of the accumulator. The problem we want to solve is computing the sum of all numbers that occur in a tree. The accumulator `sum` in Figure 4 solves this problem. Note that in equation `[1]` variable `N1` represents the current node (a number), while variable `N2` represents the sum that has been accumulated so far (also a number).

Multiply by position in tree. The last example shows the use of the

```
module Tree-pos
imports Tree-syntax
exports
context-free syntax
  pos(TREE, NAT) -> TREE # NAT
                  {traversal(accu, trafo)}
equations
[1] pos(N1, N2) = <N1 * N2, N2 + 1>
```

in	out
`pos( f( g( f(1,2), 3 ), g( g(4,5), 6 )), 0)`	`<f( g( f(0,2), 6 ), g( g(12,20), 30 )), 6>`

Figure 5: The accumulating transformer `pos` multiplies each number by its position in the tree.

accumulating transformer. The traversal functions is to determine the position of each number in a bottom-up traversal of the tree and to multiply each number by its position. This is achieved by the accumulating transformer `pos` shown in Figure 5. The general idea is to accumulate the position of each number during the traversal and to use it as a multiplier to transform numeric nodes.

4 Operational Semantics

Algorithm 2 An extended interpreter for innermost rewriting.

```
funct innermost(term, rules) ≡
  (fn, children) := decompose(term)
  children' := nil
  foreach child in children do
    children' := append(children', innermost(child, rules))
  od
  term := compose(fn, children')
  reduct := switch function-type(fn)
              case traversal : traverse(term, rules)
              case normal : reduce(term, rules);
  return if reduct = fail then term else reduct fi.
```

In this section we show an operational semantics for traversal functions. The reader is referred to [7] for a different style of semantics, namely expressing traversal functions in terms of normal rewriting systems. The semantics in this section is better suited as a reference for implementation.

We start with normal innermost rewriting as depicted earlier in Algorithm 1. In this algorithm we do assume we have a typed term representation, because we want to have *type-safe* traversal functions. This means that with every constructor and with every function symbol a first order type can be associated. For example, a function f could have type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau_r$. If $n = 0$, f is a constant of type τ_r. If $n > 0$, f is either a constructor or a function with its arguments typed by $\tau_1, ..., \tau_n$ respectively. If we allow tupled terms, we should also define the types of tuples. A tuple type is simply denoted by $(\tau_1 \times \tau_2)$. Of course, the *match* function should take care to match only terms that have matching types. Our *compose* function gets an extra argument denoting which type of function it should use to construct a term.

The main algorithm consists of the normal innermost reduction strategy, but we switch to a different mode when a traversal function is encountered. The switch statement in Algorithm 2 detects a traversal function and turns over control to a function called *traverse*, instead of calling *reduce*.

This function is shown in Algorithm 3. It initiates the traversal with different parameters for each kind of traversal function. Recall that the input term is of the form $trfn(T_1, T_2, ..., T_n)$ $(n >= 1)$ where *trfn* is a traversal function, T_1 is the term to be traversed, T_2 is the (optional) accumulator argument, and $T_3, ..., T_n$ are the (optional) remaining arguments. Actual traversal is done by *td-or-bu* ("top-down or bottom-up") that uses either *top-down* or *bottom-up* depending on the traversal strategy of *trfn*. The arguments of *td-or-bu* are determined by the different kinds of traversals.

We apply the traversal function by reusing the *reduce* function from the basic innermost rewriting algorithm. It is applied either before or after traversing the children, depending on the traversal strategy (`bottom-up` or `top-down`).

In order to be type-safe, the type of the traversal function changes while the term is traversed. Its type always matches the type of the node that is currently visited. This behavior is encoded by the *type-compose* function. For *type-preserving* transformers, the type of the first argument and the resulting argument are adapted to the type of the node that is currently visited. Similarly, for the accumulator only the type of the first argument is changed while the type of the accumulated argument remains equal. Finally, for the combination the type of the first argument and the resulting tuple type are updated.

The traversal of children in function *visit-children* takes into account that the accumulated value must be passed on between each child. Note that in case of a transformer, this accumulated value is ignored by passing always the value **nil**.

After a successful application of a user-defined rule, the function *make-reduct* decides what to do with the reduct depending on the type of traversal. If it is a transformer, the reduct replaces the redex. In case of an accumulator, the reduct replaces the accumulated argument. If it is an accumulating transformer, the first element of the tuple replaces the redex, while the second element replaces the accumulated argument.

Finally, when we return from the traversal, the top level function *traverse* returns a different normal form for each type of traversal function. In case of a transformer, the transformed term is simply returned. When it is an accumulator, we return the accumulated argument. An accumulating transformer returns a tuple of the transformed term and the final value of the accumulated argument.

Algorithm 3 An interpreter for traversal functions (part 1/2).

```
funct traverse(term, rules) ≡
  (trfn, args) := decompose(term);
  term := head(args); args := tail(args)
  switch traversal-kind(trfn)
    case "trafo" :
        (reduct, nil) := td-or-bu(trfn, term, nil, args, rules)
        return reduct
    case "accu" :
        (reduct, accu) := td-or-bu(trfn, term, head(args), tail(args), rules)
        return accu
    case "accu, trafo" :
        return td-or-bu(trfn, term, head(args), tail(args), rules); .
funct td-or-bu(trfn, term, accu, args, rules) ≡
  return switch traversal-strategy(trfn)
           case "top-down" : top-down(trfn, term, accu, args, rules)
           case "bottom-up" : bottom-up(trfn, term, accu, args, rules); .
funct top-down(trfn, term, accu, args, rules) ≡
  reduct := reduce(type-compose(trfn, term, accu, args), rules)
  return if reduct = fail then
           visit-children(trfn, term, accu, args, rules)
         else make-reduct(trfn, term, reduct) fi.
funct bottom-up(trfn, term, accu, args, rules) ≡
  (term, accu) := visit-children(trfn, term, accu, args, rules)
  reduct := reduce(type-compose(trfn, term, accu, args), rules)
  return if reduct = fail then (term, accu)
         else make-reduct(trfn, term, reduct) fi.
funct visit-children(trfn, term, accu, args, rules) ≡
  (fn, children) := decompose(term)
  children' := nil
  foreach child in children do
    (reduct, accu) := td-or-bu(trfn, child, accu, args, rules)
    children' := append(children', reduct)
  od
  return (compose(fn, children'), accu).
funct make-reduct(trfn, term, reduct) ≡
  return switch traversal-kind(trfn)
           case "trafo" : (reduct, nil)
           case "accu" : (term, reduct)
           case "accu, trafo" : reduct; .
```

5 Experience

Various experiments have been carried out including transformations on COBOL, an SDF checker, SDF re-factoring, and Java re-factoring. We discuss here the COBOL example in more detail.

In a joint project of the Software Improvement Group (SIG), Centrum voor Wiskunde en Informatica (CWI) and Vrije Universiteit (VU) traversal functions have been applied to the conversion of COBOL programs [18]. This is based on earlier work described in [15]. The purpose was to migrate VS COBOL II to COBOL/390. An existing tool (CCCA from IBM) was used to carry out the basic, technically necessary, conversions. However, this leaves

Algorithm 3 An interpreter for traversal functions (part 2/2).

funct *type-compose*(*fn*, *term*, *accu*, *args*) $\equiv$
  $\tau_t := \mathit{result\text{-}type\text{-}of}(\mathit{term})$
  switch *traversal-kind*(*fn*)
    case "`trafo`" :
      $\tau_1 \times \cdots \times \tau_n \to \tau_1 := \mathit{type\text{-}of}(\mathit{fn})$
      $\mathit{type} := \tau_t \times \cdots \times \tau_n \to \tau_t$
    case "`accu`" :
      $\tau_1 \times \tau_2 \times \cdots \times \tau_n \to \tau_2 := \mathit{type\text{-}of}(\mathit{fn})$
      $\mathit{type} := \tau_t \times \tau_2 \times \cdots \times \tau_n \to \tau_2$
    case "`accu, trafo`" :
      $\tau_1 \times \tau_2 \times \cdots \times \tau_n \to (\tau_1 \times \tau_2) := \mathit{type\text{-}of}(\mathit{fn})$
      $\mathit{type} := \tau_t \times \tau_2 \times \cdots \times \tau_n \to (\tau_t \times \tau_2)$;
  return *compose*(*fn*, *type*, [*term*, *accu*, *args*]).

many constructions unchanged that will obtain the status "archaic" or "obsolete" in the next COBOL standard. In addition, compiler-specific COBOL extensions remain in the code and several outdated run-time utilities can be replaced by standard COBOL features.

This collection of source-to-source transformations were formalized as a number of traversal functions. Every function performs a tiny sub-task. Examples of such sub-tasks are:

- Adding `END-IF` keywords to close `IF`-statements.
- Replace nested `IF`-statements with `EVALUATE`-statements.
- Replace outdated `CALL` utilities by standard COBOL statements.
- Reduce `GO-TO` statements: a goto-elimination algorithm that itself consists of over 20 different transformation rules that are applied iteratively in a fixed-point computation.

Each of these sub-tasks (21 in total) only consists of a few rewrite rules that define the effect of a traversal function on specific statements.

There are several ways of combining these sub-tasks to obtain the COBOL transformation tool that we wanted. Firstly, the sub-tasks can be combined using normal ASF+SDF rules using functional application.

Secondly, each sub-task is an ASF+SDF specification in itself, which can be run on the command-line separately. Combined with a generated parser and a generic pretty printer, such tools can be used to build complete source-to-source transformations tools using simple shell scripts.

In an experiment the transformation tools were applied to a test base of 582 COBOL programs containing 440,000 lines of code to obtain the following results:

- 17,000 `END-IF`s were added.
- 4,000 lines were changed in order to eliminate `CALL`-utilities.
- 1,000 `GO-TO`s have been eliminated (about 65% of all `GO-TO`s).
- The complete transformation took two and a half hours using the ASF

interpreter. The compiled version of traversal functions was not yet ready at the time this experiment was done but it would reduce the time by a factor of at least 15. The estimated compiled execution time would therefore be circa 10 minutes.

The number of productions in the COBOL grammar that has been used in this renovation factory was 600 and the number of transformations performed was 21 as we have seen above. Per transformation step only one traversal over the tree is needed. We can then conclude that the maximum number[4] of rewrite rules that would be needed is $21 \times 600 = 12.600$! In reality the actual number of rewrite was less than 100 thanks to the use of traversal functions.

In other projects the experience with traversal functions seems to be very positive as well. To quote [14]:

> "At the time of writing, the Framework for SDF Transformations (FST) is described by 24 traversal functions with only a few rewrite rules per function. The SDF grammar itself has about 100 relevant productions. This is a remarkable indication for the usefulness of the support for traversal functions. In worst case, we would have to deal with about 2400 rewrite rules otherwise."

6 Discussion

Traversal functions vs. hand-written code The approach sketched in this paper has substantial advantages over manually writing functions that traverse the tree explicitly.

In contrast with hand-written traversal code, the traversal functions approach is more language-independent, type safe and concise. Language-independence is obtained since the implementation of traversal functions is independent of the source language to which they are being applied. The approach is type safe since only type preserving transformations and limited sort changing transformations can be expressed. It is concise since only transformation rules have to be written for node types that actually require transformation (as opposed to nodes that only have to be visited).

In the case of hand-written code, the programmer has to traverse all nodes in the tree explicitly and the amount of code will become bulky. In addition, this code is dependent on the source language being transformed. Type safety depends on the data structure being used to represent trees. One can either use a generic tree data-type and loose type safety, or one can use distinct datatypes for each node type. In the latter case a substantial amount of interfacing and traversal code has to be written. That code lends itself, in principle, to automatic generation. For JAVA, such an approach is described in [12].

Related Work We distinguish four directly related approaches in Figure 6

[4] In practice, this number could be smaller since a hand-written specification could explicitly avoid visits to certain sub-trees.

	Untyped	Typed
Strategy primitives	Stratego [16]	ELAN [2]
Built-in strategies	Renovation Factories [8]	Traversal Functions

Figure 6: Classification of traversal approaches.

and discuss them below. For a more extensive coverage of related work we refer to [7].

ELAN [2] is a language of many-sorted, first-order, rewrite rules extended with a strategy language that controls the application of individual rewrite rules. Its strategy primitives (e.g., don't know choice, don't care choice) allow formulating non-deterministic computations. Currently, ELAN does not support generic tree traversals since they are not easily fitted in with ELAN's type system.

Stratego [17] is an untyped term rewriting language that provides user-defined strategies. Among its strategy primitives are rewrite rules and several generic traversal operators that allow the definition of any tree traversal, such as bottom-up and top-down in an abstract manner. Therefore, tree traversals are first class objects that can be reused separately from rewrite rules. Stratego provides a library with all kinds of named traversal strategies such as, for instance, `bottomup(s)`, `topdown(s)` and `innermost(s)`. In [13], a proposal is made for a Stratego-like language that allows typed generic traversals for type preserving and type-unifying strategies.

Transformation Factories [8] are an approach in which ASF+SDF rewrite rules are generated from language definitions. After the generation phase, the user instantiates an actual transformation by providing the name of the transformation and by updating default traversal behavior. Note that the generated rewrite rules are well-typed, but very unspecific types are used to obtain reusability of the generated rewrite rules.

Transformation Factories provide two kinds of traversals: transformers and analyzers. A transformer transforms the node it visits. An analyzer is the combination of a traversal, a combination function and a default value. The generated traversal function reduces each node to the default value, unless the user overrides it. The combination function combines the results in an innermost manner. The simulation of higher-order behavior again leads to unspecific types.

Relation with Traversal Functions Traversal functions emerged from our experience in writing program transformations for real-life languages in ASF+SDF. Both Stratego and Transformation Factories also offer solutions to remedy the problems that we encountered.

Stratego extends term rewriting with traversal strategy combinators and user-defined strategies, but we are more conservative and extend first-order term rewriting only with a fixed set of traversal primitives. One contribution of traversal functions is that they provide a simple *type-safe* approach for tree traversals in first-order specifications. The result is simple, can be statically type-checked in a trivial manner and can be implemented efficiently.

Recently, in [13] another type system for tree traversals was proposed. It is based on traversal combinators as found in Stratego. While this typing system is attractive in many ways, it is a bit more complicated for the user. Two generic types are added to a first-order type system: Type-Preserving and Type-Unifying strategies. To mediate between these generic types and normal types an extra combinator is offered that combines both a type-guard and a type lifting operator. In the case of traversal functions, extending the type system is not needed because the tree traversal is combined with function application in a single traversal function. This allows the interpreter or compiler to create type-guards automatically. In similar way as for traversal functions, in [13] traversal types are also divided into type-preserving effects and mappings to a single type. The tupled combination is not offered.

Compared to Transformation Factories (which most directly inspired our traversal functions), we provide a slightly different set of traversal functions and reduce the notational overhead. We have also removed the need for higher order arguments, obtaining precise and simple types. At the level of the implementation, we do not generate ASF+SDF rules, but we have incorporated traversal functions in the standard interpreter and compiler of ASF+SDF. As a result, execution is more efficient and specifications are more readable, since users are not confronted with generated rewrite rules or simulated higher-order arguments.

Compilation of Traversal Functions We have extended the ASF+SDF compiler [4] with traversal functions in a very simple manner. We only briefly sketch the approach here. For every defining rewrite rule of a traversal function and for every call to a traversal function the type of the overloaded argument and optionally the result type is turned into a single universal type. In a first approximation, the result is one big type-unsafe function which can be compiled using the existing compilation scheme.

Type-safety is achieved as follows. If the first argument of a defining rewrite rule is guarded by a constructor, this will automatically enforce type-safety. If it is not guarded (i.e., the argument is a single variable), we add a condition to the rewrite rule that checks the type of the matched tree at run-time. The compiled code will thus be type-safe.

The resulting compiled code is extended with calls to a small run-time library. They take care of actually traversing the tree and passing along the accumulated argument before or after trying to apply the compiled traversal function.

Conclusions We have described term rewriting with traversal functions as an extension to innermost term rewriting and we have shown how to incorporate this in ASF+SDF. The advantages of our approach are:

- The most frequently used traversal orders are provided as built-in primitives.
- The approach is fully type-safe and easily type-checked.

- Traversal functions can be implemented efficiently.

To summarize, traversal functions are a nice compromise between simplicity and expressive power. For implementation issues and extensive examples we refer to [7].

The main disadvantage of our approach manifests itself when dealing with traversal orders that are not provided by the built-in primitives. Two escapes are possible: such traversals can either be simulated as a modification of one of the built-in strategies (by adding conditions or auxiliary functions), or one can fall back to the tedious specification of the traversal by enumerating traversal rules for all constructors of the grammar. Extending the approach to cover the variability of left-to-right vs. right-to-left traversal is possible.

Acknowledgement

We received indispensable feedback from the users of traversal functions. Steven Klusener (Software improvement Group) and Hans Zaadnoordijk (University of Amsterdam) used them for COBOL transformations, and Ralf Lämmel (CWI and Vrije Universiteit Amsterdam) and Guido Wachsmuth (University of Rostock) applied them in SDF re-factoring.

References

[1] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1998.

[3] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[4] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 2002. To appear.

[5] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[6] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications, WRLA 98*, 1998.

[7] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36:209–266, 2000.

[9] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[10] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[11] J.W. Klop. Term rewriting systems. In D. Gabbay, S.Abramski, and T. Maibaum, editors, *Handbook of Logic and Computer Science*, volume 1. Oxford University Press, 1992.

[12] T. Kuipers and J. Visser. Object-oriented tree traversal with jjforester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[13] R. Lämmel. Typed Generic Traversals in S'_γ. Technical Report SEN-R0122, CWI, August 2001.

[14] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M.G.J. van den Brand and D. Parigot, editors, *Proc. LDTA'01*, volume 44-2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[15] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In *Proceedings of Conference on Maintenance and Reengineering (CSMR'99)*, Amsterdam, March 1999.

[16] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science. Springer-Verlag, May 2001.

[17] E. Visser, Zine-el-Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[18] H. Zaadnoordijk. Source code transformations using the new ASF+SDF meta-environment. Master's thesis, University of Amsterdam, Programming Research Group, 2001.

The Sketch of a Polymorphic Symphony

Ralf Lämmel

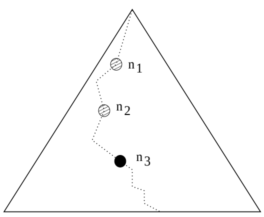
Email: `Ralf.Laemmel@cwi.nl`
CWI, Kruislaan 413, NL-1098 SJ Amsterdam
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam

Abstract

We capture common traversal schemes as typed and highly parameterised generic functions. The resulting portfolio of traversal schemes can be regarded as a benchmark for setups for generic programming. We study two original forms of parameterisation. Firstly, we design parameters serving as place holders for the specific control-flow, data-flow and traversal characteristics of more concrete traversal schemes. Secondly, we use overloading to postpone commitment to a specific type scheme for traversal. Our development is based on functional strategies, that is, first-class generic functions which can traverse into terms while mixing uniform and type-specific behaviour. Our development includes a reconstruction of functional strategies which is more compact and more general than previous models. Compactness is documented by the fact that we only rely on two somewhat special combinators. Generality is achieved via the identification of a fundamental combinator for term traversal. Our development is spelled out in Haskell 98 extended with first-class polymorphism.

Arrangement

A traversal problem Given is a tree, say a term. We want to operate on a certain node in the tree. We search for the node in some traversal order, be it top-down and left-to-right. We want to further constrain the node in the sense that it should occur on a certain path, namely below k nodes the fitness of which is determined by predicates. Once we found the node n_{k+1} below $n_1, \ldots, n_k$ we want to process the subtree t rooted by n_{k+1}. We either want to select t as is, *or* we want to compute some value from t, *or* we want to transform t in its context in the complete tree. This parameterised traversal problem is illustrated for $k = 2$ in Figure 1.



Find n_3 below n_2 in turn below n_1. We assume two predicates p_1 and p_2 to identify n_1 and n_2. We further assume a generic function f to process n_3.

Figure 1. A node on a path in a tree. Let us find it and process it!

This sort of problem is very common in programming over tree and graph structures, e.g., in adaptive object-oriented programming [19]. One usually deals with further forms of conditions, e.g., certain kinds of nodes that should *not* be passed, or further forms of computation, e.g., *cumulative* computation along the path.

Side conditions We assume, that (i) we have to cope with *many sorts*, say *systems of named, mutually recursive datatypes* such as syntaxes and formats. We have to be able to process terms of any type. The subterms of a term are potentially of different types. We further require that (ii) traversal schemes are *statically typesafe*, that is, a traversal always delivers a type-correct result (if any) without even attempting the construction of any ill-typed term. Furthermore, we insist on (iii) a *parameterised* solution where the overall traversal scheme is strictly separated from the specific predicates for the nodes $n_1, \ldots, n_k$ and the node-processing function for n_{k+1}. Also, the kind of node processing (recall selection vs. computation vs. transformation) is not anticipated. Last but not least, we require that (iv) a concise combinator approach is adopted where one can easily compose and refine traversal schemes. As for the posed problem, a faithful solution should be a concise and suggestive one-liner. With combinators, we can easily compose traversal schemes from their conceptual ingredients, e.g., options such as top-down vs. bottom-up, left-to-right vs. right-to-left, first match vs. all matches, and so on.

Conducted by Strafunski If *any* of the side conditions is given up, a solution to the posed problem becomes less satisfying. If you do not insist on static type safety, then you can resort to Prolog, or preferably to Stratego [28] which supports at least limited type checks. Types are however very desirable to guide the instantiation of traversal schemes. If you even want to do away with parameterisation and combinator style, then XSLT is your language of choice. If types appeal to you, then you might feel tempted to consider polytypic functional programming [11]. Corresponding language designs are however limited regarding parameterisation and combinator style (see [17] for details). If you are as demanding as we are, then you use the *Strafunski* approach to generic programming in Haskell.[1] In [16], Joost Visser and the present author developed a corresponding notion of functional strategies, that is, first-class generic functions which can traverse into terms while mixing uniform and type-specific behaviour. Functional strategies are first-class citizens because construction, composition, and customisation of strategies is based on generic function combinators. The term "strategy" hints on strategic term rewriting [28] to honour some important ideas underlying functional strategies. The foundations of *typeful* strategic term rewriting were worked out in [14].

The movements No previous knowledge of strategic programming is required to enjoy this symphony. We reconstruct functional strategies as an amalgamation of certain forms of polymorphism. The resulting model of functional strategies is much more compact and more general than in our previous work [15,16]. This polymorphic symphony is brought to you by Haskell 98 [23] extended with first-

[1] *Strafunski* home page: `http://www.cs.vu.nl/Strafunski` where `Stra` refers to strategies, `fun` refers to functional programming, and their harmonious composition is a homage to the music of Igor Stravinsky.

class polymorphism [13]. The symphony consists of four movements. In Section 1, the type *Generic* that models functional strategies is initiated. We define parametrically polymorphic [25,29] combinators on *Generic*. In Section 2, we integrate a combinator *adhoc* to update functional strategies by monomorphic ad-hoc cases. This combinator relies on type case—a notion that was studied, for example, in the context of dynamic typing [1,2,26] and intensional polymorphism [8,32]. In Section 3, we integrate a fundamental combinator *hfoldr* to perform single steps of traversal. This combinator folds over the immediate subterms of a term, but it does not recurse by itself. This kind of fold can be regarded as a variation on polytypism [11]. In Section 4, we use overloading, say ad-hoc polymorphism [31,12], to treat different types of traversal (recall computation vs. transformation) in a uniform manner. Ultimately, we define a portfolio of highly parameterised traversal schemes, including the one for the introductory traversal problem. This portfolio demonstrates the expressiveness and conciseness of functional strategies.

Availability The code shown in the paper is fully operational in Haskell (tested with hugs98; Dec. 2001 version). A paper-specific *Strafunski* distribution with illustrative examples can be downloaded from the paper's web-site. [2] This distribution also comes with generative tool support for user-supplied datatypes (say, syntaxes or formats) so that functional strategies can be used for these datatypes.

1 Moderato

In this section, we initiate the type of generic functions that model functional strategies. For the sake of a systematic development starting from standard concepts, we will first focus on the parametrically polymorphic dimension of the corresponding type scheme. The subsequent two movements add expressiveness so that functional strategies can go beyond parametric polymorphism. This additional expressiveness will have to do with term traversal and with type-specific customisation.

First-class polymorphic functions In Figure 2, we use a type synonym *Parametric* with two parameters α and κ to capture the general type scheme of functional strategies, that is $\alpha \rightarrow \kappa\ \alpha$. As one can see, the parameter α corresponds to the domain of the function type, and κ is used to construct the co-domain from α. Common options for the co-domain type constructor κ are the identity type constructor I and the constant type constructor C as defined in the figure, too. Based on the type scheme *Parametric*, we define the ultimate datatype *Generic* to capture generic functions which are universally quantified in α. This definition involves a class constraint "*Term* $\alpha\ \Rightarrow \ldots$" to prepare for the later movements which go beyond parametric polymorphism. One should neglect the constraint for a while. But note that the function type with the "$\forall$" occurs as a *component* of the datatype constructor G. This form of (second-order) polymorphism, where polymorphic entities are wrapped by datatype constructors, is called first-class polymorphism [13], and it is supported as a common language extension of Haskell 98. (Implicit) top-level universal quantification, as favoured by pure Haskell 98, would be eventually insufficient to construct generic function combinators.

[2] `http://www.cwi.nl/~ralf/polymorphic-symphony/`

The generic function type

$\mathbf{type}\ \mathit{Parametric}\ \alpha\ \kappa = \alpha \to \kappa\ \alpha$
$\mathbf{newtype}\ \mathit{Generic}\ \kappa = G\ (\forall\ \alpha.\ \mathit{Term}\ \alpha \Rightarrow \mathit{Parametric}\ \alpha\ \kappa)$
$\mathit{unG}\ (G\ f) = f$

The identity type constructor

$\mathbf{newtype}\ I\ \alpha = I\ \alpha$
$\mathit{unI}\ (I\ x) = x$

The constant type constructor

$\mathbf{newtype}\ C\ u\ \alpha = C\ u$
$\mathit{unC}\ (C\ x) = x$

Sequential type constructor composition

$\mathbf{newtype}\ S\ t\ t'\ \alpha = S\ (t'\ (t\ \alpha))$
$\mathit{unS}\ (S\ x) = x$

Type-preserving and type-unifying strategies

$\mathbf{type}\ \mathit{MG}\ \kappa\ m = \mathit{Generic}\ (S\ \kappa\ m)$ -- Monadic generic functions
$\mathbf{type}\ \mathit{TP}\ m = \mathit{MG}\ I\ m$ -- Corresponds to $\forall\alpha.\ \alpha \to m\ \alpha$
$\mathbf{type}\ \mathit{TU}\ u\ m = \mathit{MG}\ (C\ u)\ m$ -- Corresponds to $\forall\alpha.\ \alpha \to m\ u$

Strategy application by unwrapping

$\mathit{apply} :: (\mathit{Term}\ \alpha, \mathit{Monad}\ m) \Rightarrow \mathit{MG}\ \kappa\ m \to \alpha \to m\ (\kappa\ \alpha)$
$\mathit{apply}\ f\ x = \mathit{unS}\ (\mathit{unG}\ f\ x)$

More convenient application for *TP*

$\mathit{applyTP} :: (\mathit{Monad}\ m, \mathit{Term}\ \alpha) \Rightarrow \mathit{TP}\ m \to \alpha \to m\ \alpha$
$\mathit{applyTP}\ s\ x = \mathit{apply}\ s\ x \ggg\!= \mathit{return} \circ \mathit{unI}$

More convenient application for *TU*

$\mathit{applyTU} :: (\mathit{Monad}\ m, \mathit{Term}\ \alpha) \Rightarrow \mathit{TU}\ u\ m \to \alpha \to m\ u$
$\mathit{applyTU}\ s\ x = \mathit{apply}\ s\ x \ggg\!= \mathit{return} \circ \mathit{unC}$

Figure 2. Functional strategies as first-class polymorphic functions

Besides the identity and constant type constructors I and C, we also define a constructor S for (sequential) composition of type constructors in the figure. Based on these constructors, we identify three classes of generic functions in terms of *Generic* in the figure. The synonym *MG* captures monadic generic functions. It is convenient to assume that functional strategies are monadic [30] because that way we can deal with effects during traversal, e.g., success and failure, state, or nondeterminism. The synonym *TP* instantiates *MG* via I so that *T*ype-*P*reserving strategies are identified. The synonym *TU* instantiates *MG* via C so that *T*ype-*U*nifying strategies are identified, that is, functions with a fixed result type u.

We complete the development in the figure with trivial definitions of combinators for strategy application. The combinator *apply* performs some unwrapping prior to ordinary function application. Unwrapping accounts for the style of first-class polymorphism (cf. *unG*), and for type-constructor composition (cf. *unS*). For convenience, the combinators *applyTP* and *applyTU* specialise *apply* for *TP* and *TU* to hide the employment of datatypes for the co-domain construction. Their result types point out the basic type schemes for *TP* and *TU* without any noise.

Parametrically polymorphic embedding

$$para :: (\forall\, \alpha.\ Parametric\ \alpha\ \kappa \rightarrow Generic\ \kappa)$$
$$para\ f = G\ f$$

Identity strategy

$$idTP :: Monad\ m \Rightarrow TP\ m$$
$$idTP = para\ (S \circ return \circ I)$$

Constant strategy

$$constTU :: Monad\ m \Rightarrow u \rightarrow TU\ u\ m$$
$$constTU\ u = para\ (S \circ return \circ C \circ const\ u)$$

Failure strategy

$$fail :: MonadPlus\ m \Rightarrow MG\ \kappa\ m$$
$$fail = para\ (S \circ const\ mzero)$$

Figure 3. Parametrically polymorphic strategies

Nullary combinators Let us start to inhabit the strategy types. So far we can only provide parametrically polymorphic inhabitants [25,29]. In Section 2 and Section 3, we will inhabit the type *Generic* in other ways by employing the "beyond parametric polymorphism" interface of the *Term* class. If we neglect the monadic facet of functional strategies for a second, then there are just the following parametrically polymorphic inhabitants: the identity function for the type-preserving scheme, and constant functions for the type-unifying scheme. This generalises accordingly for the monadic setup as spelled out in Figure 3. For clarity, we define the function *para* which models the approval of a parametrically polymorphic function as a member of *Generic*. Its definition coincides with the constructor G but its type insists on a parametrically polymorphic argument without mentioning the *Term* class. We can construct the identity and constant strategies as denoted by *idTP* and *constTU*. Relying on the extended monad class *MonadPlus*, we can also define the always failing strategy *fail* in terms of the member *mzero* denoting failure, e.g., *Nothing* in the case of the *Maybe* monad. This last inhabitant points out that functional strategies might exhibit success and failure behaviour, or they even might be non-deterministic depending on the actual choice of the monad [30].

Binary combinators In Figure 4, we define three prime combinators to compose functional strategies. The combinator *seq* lifts monadic sequencing of function applications to the strategy level (cf. "$\gg=$"). The combinator *pass* composes two strategies which share the term argument and where the result of the first strategy is passed as additional input to the second strategy. The combinator *choice* composes alternative function applications. The actual kind of choice depends on the (extended) monad which is employed in the definition (cf. class *MonadPlus* and its member *mplus*), e.g., the *Maybe* monad for partiality or the *List* monad for multiple results. In the definitions of the binary combinators, we use *apply* to deploy the polymorphic function arguments, and we use G to re-wrap the composition as a polymorphic function of type *Generic*. As an aside, we cannot use *para* instead of G because eventually, we want to compose functions that go beyond parametric polymorphism.

Left-to-right monadic sequential strategy composition

$seq :: Monad\ m \Rightarrow TP\ m \rightarrow MG\ \kappa\ m \rightarrow MG\ \kappa\ m$

$seq\ f\ g = G\ (\lambda x \rightarrow S\ (applyTP\ f\ x \gg\!= apply\ g))$

Value-passing with shared term argument

$pass :: Monad\ m \Rightarrow TU\ u\ m \rightarrow (u \rightarrow MG\ \kappa\ m) \rightarrow MG\ \kappa\ m$

$pass\ f\ g = G\ (\lambda x \rightarrow S\ (applyTU\ f\ x \gg\!= \lambda u \rightarrow apply\ (g\ u)\ x))$

Composition of alternative strategies

$choice :: MonadPlus\ m \Rightarrow MG\ \kappa\ m \rightarrow MG\ \kappa\ m \rightarrow MG\ \kappa\ m$

$choice\ f\ g = G\ (\lambda x \rightarrow S\ (apply\ f\ x\ \text{`}mplus\text{`}\ apply\ g\ x))$

Figure 4. Composition combinators for functional strategies

Rank-2 types A crucial observation is that the composition combinators *seq*, *pass* and *choice* really enforce us to use second-order polymorphism [7,24]. Standard function combinators like "$\circ$" or "$\gg\!=$" are polymorphic but these combinators have rank-1 types, that is, they are quantified at the top-level. This implies monomorphic function arguments. By contrast, the types of *seq*, *pass* and *choice* are quantified argument-wise. These rank-2 types model that the combinators operate on polymorphic functions as required for generic programming in combinator style. The traversal combinators and schemes in the following sections have rank-2 types, too. Instead of using first-class polymorphism, we could consider plain rank-2 types as meanwhile available in the GHC implementation of Haskell. We can then directly deal with polymorphic functions as opposed to wrapped first-class polymorphic functions. On the other hand, some amount of type annotations will then be mandatory [33,22]. We investigate such a model in [17]. Note that a further sophistication of Haskell would be needed to eliminate the other main source of noise in our encoding of functional strategies. We needed type-level lambdas to use proper type constructors instead of datatypes such as I, C, and S.

2 Larghetto

At this point in our polymorphic concert, we still lack the original expressiveness of functional strategies: the ability to perform traversal into terms while mixing uniform and type-specific behaviour. This section and the subsequent one are concerned with this strategic expressiveness. In this section, we provide a distinguished combinator *adhoc* for updating functional strategies by type-specific cases. In the next section, traversal into terms will be enabled.

Strategy update The type of the *adhoc* combinator is given at the top of Figure 5. The combinator allows us to impose a monomorphic function as an ad-hoc case on an existing functional strategy. We call this idiom *strategy update* because of its affinity to point-wise modification of ordinary functions. In this sense, strategy update is about "type-wise" modification of a polymorphic function. Here are two illustrative examples of strategies that involve an ad-hoc case:

$$\begin{aligned} negbool &:: TP\ Identity \\ negbool &= adhocTP\ idTP\ (return \circ \neg) \end{aligned} \qquad \begin{aligned} recint &:: TU\ Int\ Maybe \\ recint &= adhocTU\ fail\ return \end{aligned}$$

The *adhoc* combinator

$adhoc :: Term\ \alpha \Rightarrow Generic\ \kappa \rightarrow Parametric\ \alpha\ \kappa \rightarrow Generic\ \kappa$
$adhoc\ poly\ mono = ...$ -- Implemented via *Term* class

A convenient specialization for *TP*

$adhocTP :: (Monad\ m, Term\ \alpha) \Rightarrow TP\ m \rightarrow (\alpha \rightarrow m\ \alpha) \rightarrow TP\ m$
$adhocTP\ poly\ mono = adhoc\ poly\ (\lambda x \rightarrow S\ (mono\ x \ggg= return \circ I))$

A convenient specialization for *TU*

$adhocTU :: (Monad\ m, Term\ \alpha) \Rightarrow TU\ u\ m \rightarrow (\alpha \rightarrow m\ u) \rightarrow TU\ u\ m$
$adhocTU\ poly\ mono = adhoc\ poly\ (\lambda x \rightarrow S\ (mono\ x \ggg= return \circ C))$

Figure 5. Updating functional strategies by type-specific cases

The strategy *negbool* behaves like *idTP* most of the time but it performs negation when faced with a Boolean (cf. "¬"). We use the *Identity* monad. The strategy *recint* is meant to recognise integers, that is, it simply fails if it does not find an integer. We use the *Maybe* monad to deal with failure. For convenience, we use the specialisations *adhocTP* and *adhocTU* from Figure 5 which allow the strategic programmer to forget about wrapping and unwrapping I, C, and S.

Strategy update is defined as follows. Given a polymorphic function *poly* of type $Generic\ \kappa$, and a monomorphic function *mono* of type $Parametric\ \alpha\ \kappa$, the updated function $adhoc\ poly\ mono$ branches according to the type α' of an eventual input value v: if $\alpha = \alpha'$, then *mono* is applied to v, otherwise *poly*. That is:

$$adhoc\ poly\ mono\ v\ = \begin{cases} mono\ v, & \text{if type of } v = \text{domain of } mono \\ poly\ v, & \text{otherwise} \end{cases}$$

A strategic programmer considers *adhoc* as a primitive. In Haskell, we cannot define the *adhoc* combinator once and for all, but the combinator can be easily supported per datatype (see the Appendix and [16,17]). This is precisely what the *Term* class in the definition of *Generic* is needed for (recall Figure 2). Strategy update can be considered as a disciplined form of type case as it was investigated, for example, in the context of dynamic typing [1,2,26] and intensional polymorphism [8,32], more specifically as a form of type-based function dispatch (see [17,14,16] for more technical and bibliographical details).

Mixture of specificity and genericity The contribution of strategy update to the notion of a functional strategy is that it allows us to lift type-specific behaviour in a type-safe and transparent way to the strategy level. Note that parametric polymorphism only works the other way around: a polymorphic function can be applied to a value of a specific type, and in fact, the actual behaviour will be entirely uniform, independent of the specific type [25,29]. In generic programming with traversals, type-specific customisation is indispensable because a traversal is typically concerned with certain types in a specific manner to get its job done. Recall the introductory example. While we ask for a completely generic and uniform traversal scheme, type-specific behaviour is still due when the scheme is instantiated. The concrete predicates $p_1, \ldots, p_k$ constraining the path, and the node-processing function f for the final node will most likely involve type-specific behaviour. In Figure 6, this is demonstrated by an illustrative instantiation of the introductory

In the following instantiation scenario for Figure 1, we assume that the desired traversal scheme is called *belowlist* (to be defined later). The function *belowlist* takes a node-processing function f and a list of predicates for nodes that have to be passed before processing a node with f.

The shown Haskell code declares a sample system of mutually recursive datatypes *SortA* and *SortB*. It further defines a strategy *b2int* to identify certain *SortB* terms and to extract their *Int*eger component. We consider the following instance of the introductory traversal problem: Given a term, we want to find a node of type *SortB* to extract its *Int*eger component if it occurs on a path with two *SortB* nodes with the *Int*eger components 1 and 3. The sample term *term1* contains a node that meets all the requirements, namely a node with 42 as *Int*eger component. The traversal executes as follows in a Hugs session:

```
Main> test42
Just 42
Main>
```

A sample system of two datatypes

$$\begin{array}{l}\textbf{data}\ \mathit{SortA} = \mathit{SortA1}\ \mathit{SortB} \mid \mathit{SortA2}\\ \textbf{data}\ \mathit{SortB} = \mathit{SortB}\ \mathit{Int}\ \mathit{SortA}\end{array}$$

A test term for traversal

$$\begin{array}{l}\mathit{term1} = \mathit{SortA1}\ (\mathit{SortB}\ 0\ (\\ \qquad \mathit{SortA1}\ (\mathit{SortB}\ 1\ (\\ \qquad \mathit{SortA1}\ (\mathit{SortB}\ 2\ (\\ \qquad \mathit{SortA1}\ (\mathit{SortB}\ 3\ (\\ \qquad \mathit{SortA1}\ (\mathit{SortB}\ 42\\ \qquad \mathit{SortA2}))))))))\end{array}$$

Extracting the *Int*eger from a *SortB* term

$$\begin{array}{l}\mathit{b2int} :: \mathit{TU}\ \mathit{Int}\ \mathit{Maybe}\\ \mathit{b2int} = \mathit{adhocTU}\\ \qquad \mathit{fail}\\ \qquad (\lambda(\mathit{SortB}\ i\ \_) \rightarrow \mathit{Just}\ i)\end{array}$$

A constant filter

$$\begin{array}{l}\mathit{eqTU} :: \mathit{Eq}\ a\\ \qquad \Rightarrow \mathit{TU}\ a\ \mathit{Maybe}\\ \qquad \rightarrow a \rightarrow \mathit{TU}\ ()\ \mathit{Maybe}\\ \mathit{eqTU}\ s\ i = s\ \text{`}\mathit{pass}\text{`}\ (\lambda i' \rightarrow\\ \qquad \textbf{if}\ i' \equiv i\ \textbf{then}\ \mathit{constTU}\ ()\\ \qquad\qquad \textbf{else}\ \mathit{fail})\end{array}$$

The actual scheme instantiation

$$\begin{array}{l}\mathit{test42} :: \mathit{Maybe}\ \mathit{Int}\\ \mathit{test42} = \mathit{applyTU}\ (f\ \text{`}\mathit{belowlist}\text{`}\ [\mathit{p1}, \mathit{p2}])\\ \qquad\qquad \mathit{term1}\\ \quad \textbf{where}\ f = \mathit{b2int}\\ \qquad\quad \mathit{p1} = \mathit{b2int}\ \text{`}\mathit{eqTU}\text{`}\ 1\\ \qquad\quad \mathit{p2} = \mathit{b2int}\ \text{`}\mathit{eqTU}\text{`}\ 3\end{array}$$

Figure 6. Instantiation of the introductory problem

problem. At a more general level, one can say that the *adhoc* combinator implements the following admittedly very much related requirements regarding the tension between genericity and specificity:

(i) Separability of generic and type-specific behaviour

(ii) Composability of generic and type-specific behaviour

(iii) First-class status of updatable generic functions

In Haskell, the common approach to the definition of polymorphic functions with type-specific behaviour is to use ad-hoc polymorphism, say type and constructor classes [31,12]. This approach does not meet the above requirements because a class and its instances form a declaration unit. Any new combination of generic and type-specific behaviour has to be anticipated by a dedicated class. Hence, strategy update is complementary. We refer to [14,17] for a discussion of generic programming approaches in the light of mixing uniform and type-specific behaviour.

3 Allegretto

Ultimately, functional strategies are meant to perform term traversal. Hence, we need to add another bit of polymorphism. In fact, the kind of term traversal that we envisage can be regarded as a variation on polytypism [11]. That is, we want to recurse into terms on the basis of the structure of the underlying algebraic datatypes. We will first briefly categorise traversal approaches in previous work. Then we will generalise one of the approaches to contribute a very flexible means of traversal to the ongoing reconstruction of functional strategies.

Traversal approaches We identify the following categories:

(i) Recursive traversal schemes [21,5,27,18,4]

(ii) One-step traversal combinators [28,14,16]

(iii) Traversal by type induction [8,11,32]

(iv) Traversal at a representation-type level [3]

(i) Recursive traversal schemes, e.g., generalised folds or iterators over data structures, follow a fixed scheme of recursion into terms, and the style of composition of intermediate results is also intertwined with the recursion scheme. (ii) One-step traversal combinators add flexibility because they do not recurse into terms by themselves but they rather capture single steps of traversals. Hence, different recursion schemes and different means to compose intermediate results can still be established by plain (recursive) function definition. The value of traversal combinators was identified in the context of strategic term rewriting [28] in the application domain of program transformation. (iii) Intensional type analysis or polytypic programming suggests that a traversal can also be viewed as a function that is defined by type induction on the argument. Type induction does not need to be restricted to the argument of a function, neither is it restricted to term types. The corresponding type systems, language designs, and implementational models are rather involved. Still they are limited regarding a faithful encoding of functional strategies (see [17] for details). (iv) Finally, a programmer can resort to a universal representation type to perform traversal. Here we assume that the programmer is provided with implosion and explosion functionality to mediate between programmer-supplied term types and the representation type. While unrestricted access to a representation type corresponds to a less safe way of generic programming due to the potential of implosion problems, *implementations* of (i)–(iii) might very well be based on a representation type. To give an example, in [16], traversal combinators are implemented using a representation type without any exposure to the generic programmer. Hence, implosion safety can be certified.

Folding over lists of children In this symphony, we basically follow the one-step traversal-combinator approach. In fact, we generalise this approach in the following manner. Rather than making a somewhat arbitrary selection of traversal combinators as in previous work [28,14,16], we restrict ourselves to one fundamental principle, namely folding over the immediate subterms of a term (say, its children). All combinators which can be expressed by a fold over the children are then *derivable*. We contend that a fold over constructor applications is the general

Fold algebras	
$\mathbf{data}\ \mathit{HFoldrAlg}\ \kappa = A$	-- First-class polymorphic wrapper
$(\forall\, \alpha\, \beta.\ \mathit{Term}\ \alpha \Rightarrow \alpha \to \kappa\ (\alpha \to \beta) \to \kappa\ \beta)$	-- Cons-like case
$(\forall\, \gamma.\ \gamma \to \kappa\ \gamma)$	-- Nil-like case
The fold combinator	
$\mathit{hfoldr} :: \mathit{HFoldrAlg}\ \kappa \to \mathit{Generic}\ \kappa$	-- Apply a fold algebra
$\mathit{hfoldr} = ...$	-- Implemented via *Term* class

Figure 7. Folding over constructor applications

scheme underlying one-step traversal combinators. Let us characterise the fold we need. Since the children of a term form a heterogeneous list of terms, we need a *heterogeneous* fold. Without loss of generality, we favour a *right-associative* fold in the sequel. We call the resulting fold operation *hfoldr* for right-associative, heterogeneous fold. Let us recall how the ordinary *foldr* folds over a homogeneous list. Given the fold ingredients f and z for the non-empty and the empty list form, the application of *foldr* to a list is defined as follows:

$$\mathit{foldr}\ f\ z\ [x_1, x_2, \ldots, x_n]\ =\ f\ x_1\ (f\ x_2\ (\cdots\ (f\ x_n\ z)\ \cdots))$$

The combinator *hfoldr* folds over a constructor application in nearly the same way:

$$\mathit{hfoldr}\ f\ z\ (C\ x_n\ \cdots\ x_2\ x_1)\ =\ f\ x_1\ (f\ x_2\ (\cdots\ (f\ x_n\ (z\ C))\ \cdots))$$

Note the order of the children in a constructor application. Due to curried style, we have a kind of snoc-list, that is, the leftmost child is the head of the list, and so on. Also note that the empty constructor application C is passed to z. This is necessary to keep track of the specific datatype constructor. A strategic programmer considers *hfoldr* as a primitive. In Haskell, *hfoldr* needs to be instantiated per datatype, that is, we need an equation of the above form for each specific constructor (see the Appendix for an illustration).

Fold algebras In Haskell, the ingredients f and z for *hfoldr* need to be packed in a datatype, say a fold algebra, in deviation from curried style used for *foldr*. This is necessary because the ingredients of *hfoldr* rely on first-class polymorphism. The corresponding fold algebra type *HFoldrAlg* is defined in Figure 7. There is one constructor A with two components corresponding to the ingredients f and z. The first component is universally quantified in α and β corresponding to the type of the heading child (i.e., the outermost argument in the constructor application), and the type of the constructor application at hand, respectively. The type of the recursively processed tail is $\kappa\ (\alpha \to \beta)$ to witness that the tail lacks a child of type α. Given a constructor application $C\ x_n\ \cdots\ x_2\ x_1$, the type variable α would be bound to the respective types of the x_i in the several folding steps while the type β variable would be bound to the corresponding remainders of C's type. The second component of A (corresponding to the base case z) is simply a parametrically polymorphic function that is meant to process the empty constructor application. Note that the type variable γ will be typically instantiated by a function type.

One-step traversal combinators Based on *hfoldr*, one can define all kinds of one-step traversal combinators. We indicate an open-ended list of candidates by reconstructing combinators that were presented elsewhere [28,15,16,14]:

Process all children; preserve the outermost constructor

$$
\begin{aligned}
&\mathit{allTP} :: \mathit{Monad}\ m \Rightarrow \mathit{TP}\ m \rightarrow \mathit{TP}\ m \\
&\mathit{allTP}\ s = \mathit{hfoldr}\ (A\ f\ z) \\
&\quad \textbf{where} \\
&\quad\ f\ h\ t = S\ (\textbf{do}\ t' \leftarrow \mathit{unS}\ t \\
&\qquad\qquad\qquad\quad h' \leftarrow \mathit{applyTP}\ s\ h \\
&\qquad\qquad\qquad\quad \mathit{return}\ (I\ ((\mathit{unI}\ t')\ h'))) \\
&\quad\ z = S \circ \mathit{return} \circ I
\end{aligned}
$$

Try to process the children until one succeeds; preserve the outermost constructor

$$
\begin{aligned}
&\mathit{oneTP} :: \mathit{MonadPlus}\ m \Rightarrow \mathit{TP}\ m \rightarrow \mathit{TP}\ m \\
&\mathit{oneTP}\ s = G\ (\lambda x \rightarrow S\ (\mathit{apply}\ (\mathit{poneTP}\ s)\ x \ggg\!= \\
&\qquad\qquad\qquad\qquad \mathit{maybe}\ \mathit{mzero}\ (\mathit{return} \circ I) \circ \mathit{fst} \circ \mathit{unD}))
\end{aligned}
$$

The Duplicate type constructor for encoding type-preserving paramorphisms

$$
\begin{aligned}
&\textbf{newtype}\ D\ t\ \alpha = D\ (t\ \alpha, t\ \alpha) \\
&\mathit{unD}\ (D\ x) = x
\end{aligned}
$$

Paramorphic helper for *oneTP*

$$
\begin{aligned}
&\mathit{poneTP} :: \mathit{MonadPlus}\ m \Rightarrow \mathit{TP}\ m \rightarrow \mathit{MG}\ (D\ \mathit{Maybe})\ m \\
&\mathit{poneTP}\ s = \mathit{hfoldr}\ (A\ f\ z) \\
&\quad \textbf{where} \\
&\quad\ f\ h\ t = S\ (\textbf{do}\ \{t' \leftarrow \mathit{unS}\ t; \\
&\qquad\qquad\qquad\quad \mathit{tp} \leftarrow \mathit{maybe}\ \mathit{mzero}\ \mathit{return}\ (\mathit{snd}\ (\mathit{unD}\ t')); \\
&\qquad\qquad\qquad\quad (\textbf{do}\ \mathit{tc} \leftarrow \mathit{maybe}\ \mathit{mzero}\ \mathit{return}\ (\mathit{fst}\ (\mathit{unD}\ t')) \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ (D\ (\mathit{Just}\ (\mathit{tc}\ h), \mathit{Just}\ (\mathit{tp}\ h))) \\
&\qquad\qquad\qquad\quad)\ `\mathit{mplus}`\ (\\
&\qquad\qquad\qquad\quad\ \textbf{do}\ h' \leftarrow \mathit{applyTP}\ s\ h \\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ (D\ (\mathit{Just}\ (\mathit{tp}\ h'), \mathit{Just}\ (\mathit{tp}\ h))) \\
&\qquad\qquad\qquad\quad)\ `\mathit{mplus}`\ \mathit{return}\ (D\ (\mathit{Nothing}, \mathit{Just}\ (\mathit{tp}\ h)))\}) \\
&\quad\ z\ x = S\ (\mathit{return}\ (D\ (\mathit{Nothing}, \mathit{Just}\ x)))
\end{aligned}
$$

Process all children; reduce the intermediate results

$$
\begin{aligned}
&\mathit{allTU} :: \mathit{Monad}\ m \Rightarrow (u \rightarrow u \rightarrow u) \rightarrow u \rightarrow \mathit{TU}\ u\ m \rightarrow \mathit{TU}\ u\ m \\
&\mathit{allTU}\ \mathit{op2}\ \mathit{unit}\ s = \mathit{hfoldr}\ (A\ f\ z) \\
&\quad \textbf{where} \\
&\quad\ f\ h\ t = S\ (\textbf{do}\ a \leftarrow \mathit{unS}\ t \\
&\qquad\qquad\qquad\quad b \leftarrow \mathit{applyTU}\ s\ h \\
&\qquad\qquad\qquad\quad \mathit{return}\ (C\ (\mathit{unC}\ a\ `\mathit{op2}`\ b))) \\
&\quad\ z = S \circ \mathit{return} \circ C \circ \mathit{const}\ \mathit{unit}
\end{aligned}
$$

Try to process the children until one succeeds; return the processed child

$$
\begin{aligned}
&\mathit{oneTU} :: \mathit{MonadPlus}\ m \Rightarrow \mathit{TU}\ u\ m \rightarrow \mathit{TU}\ u\ m \\
&\mathit{oneTU}\ s = \mathit{hfoldr}\ (A\ f\ z) \\
&\quad \textbf{where} \\
&\quad\ f\ h\ t = S\ (\mathit{cast}\ (\mathit{unS}\ t)\ `\mathit{mplus}`\ \mathit{cast}\ (\mathit{apply}\ s\ h)) \\
&\quad\ z = S \circ \mathit{const}\ \mathit{mzero} \\
&\quad\ \mathit{cast}\ c = c \ggg\!= \mathit{return} \circ C \circ \mathit{unC}
\end{aligned}
$$

Figure 8. Examples of one-step traversal combinators

allTP Process all children; preserve the constructor.

oneTP Try to process the children until one succeeds; preserve the constructor.

allTU Process all children; reduce the intermediate results.

oneTU Try to process the children until one succeeds; return the processed child.

One can think of further combinators, e.g., combinators dealing with different orders of processing children, monadic effects, and different constraints regarding success and failure behaviour.

The definition of the combinators is shown in Figure 8. Let us explain the first one, that is, *allTP*. We construct a (type-preserving) fold algebra $A\ f\ z$ and pass it to *hfoldr*. The ingredient z for the empty constructor application resorts to *return*, that is, the outermost constructor is preserved. Additional wrapping of S and I is needed to deal with the datatypes-as-type-constructors encoding in Haskell. The ingredient f sequences three computations. Firstly, the tail t is computed. Secondly, the head h (i.e., the child at hand) is processed via s. Thirdly, the processed head h' is passed to the computed tail t' (modulo wrapping and unwrapping). The definition of the other combinators is similar. A minor complication shows up when we want to define *oneTP* because it turns out that this fold over children is paramorphic in nature [20] while our combinator *hfoldr* is catamorphic. We use a pairing technique adopted from [20] to encode the paramorphic fold with *hfoldr* (cf. the new type constructor D). This complication also illustrates the potential for further generic function types, namely $MG\ (D\ Maybe)\ m$ in this case.

Sums of products vs. curried constructor applications The combinator *hfoldr* roughly covers the sum and product cases in a polytypic definition [11]. While a polytypic definition deals with the many children in a term via nested binary products, our fold processes all children in a curried constructor application as is. While the different constructors of a datatype amount to nested binary sums in a polytypic definition, we dedicate the base case of folding to the empty constructor application. That is, polytypic definitions resort to a constructor-free representation type. Also note that polytypic definitions employ a designated declaration form for type induction while functional strategies are defined as ordinary recursive functions. Finally, polytypic programs are typically implemented by compile-time specialisation of polytypic definitions. In our setting, we only require a *Term* interface for datatypes to enable *hfoldr* and *adhoc*. A related effort in language design is described in [6] where a functional language *Rho-Stratego* is designed. This language includes a construct similar in intention to our *hfoldr*: pattern matching is generalised in a way that generic access is granted to subterms of constructor applications. *Rho-Stratego* received inspiration from higher-order rewriting while our approach is more closely related to the idioms of polytypism and type case.

4 Largo

We are now in the position to define all kinds of traversal schemes in terms of the strategy combinators and recursion. Before we present a portfolio of such schemes, we impose some extra structure on our combinator suite in order to allow for a uniform treatment of type-preserving and type-unifying traversal.

Overloading *TP* and *TU* The strategy combinators are usually typed in a manner that they can be used to derive strategies of both type *TP* and type *TU*. Hence, one would expect that traversal schemes can be composed in a manner to work for both type schemes as well. There are two asymmetries which we need to address.

Overloading operator(s) involving *MonadPlus*

$$
\begin{array}{l}
\quad \textbf{class}\ \mathit{GenericMonadPlus}\ s\ \textbf{where} \\
\qquad \mathit{one} \ :: \ s \to s \\
\quad \textbf{instance}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{GenericMonadPlus}\ (\mathit{TP}\ m)\ \textbf{where} \\
\qquad \mathit{one} \ = \ \mathit{oneTP} \\
\quad \textbf{instance}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{GenericMonadPlus}\ (\mathit{TU}\ u\ m)\ \textbf{where} \\
\qquad \mathit{one} \ = \ \mathit{oneTU}
\end{array}
$$

Overloading operators potentially involving *Monoid*

$$
\begin{array}{l}
\quad \textbf{class}\ \mathit{GenericMonoid}\ s\ \textbf{where} \\
\qquad \mathit{skip} \ :: \ s \\
\qquad \mathit{all} \ :: \ s \to s \\
\qquad \mathit{comb} \ :: \ s \to s \to s \\
\quad \textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{GenericMonoid}\ (\mathit{TP}\ m)\ \textbf{where} \\
\qquad \mathit{skip} \ = \ \mathit{idTP} \\
\qquad \mathit{all} \ = \ \mathit{allTP} \\
\qquad \mathit{comb} \ = \ \mathit{seq} \\
\quad \textbf{instance}\ (\mathit{Monad}\ m, \mathit{Monoid}\ u) \Rightarrow \mathit{GenericMonoid}\ (\mathit{TU}\ u\ m)\ \textbf{where} \\
\qquad \mathit{skip} \ = \ \mathit{constTU}\ \mathit{mempty} \\
\qquad \mathit{all} \ = \ \mathit{allTU}\ \mathit{mappend}\ \mathit{mempty} \\
\qquad \mathit{comb} \ = \ \lambda s\ s' \to s\ \text{`}\mathit{pass}\text{`}\ \lambda a \to s'\ \text{`}\mathit{pass}\text{`}\ \lambda b \to \mathit{constTU}\ (a\ \text{`}\mathit{mappend}\text{`}\ b)
\end{array}
$$

Figure 9. Classes of generic function combinators

Firstly, there is no obvious way to write fold algebras which cover both *TP* and *TU*. This is the reason why we have a *oneTP* vs. a *oneTU*, and an *allTP* vs. an *allTU*. These combinators should be overloaded. Secondly, there is no uniform way to *combine* the execution of two strategies of the *same* type. We can only compose *alternative* branches via *choice*. For the sake of a generic sequential composition, the *seq* combinator insists unavoidably on a type-preserving first argument. For the sake of value passing, the *pass* combinator needs a type-unifying first argument. Hence, we suggest an overloaded composition combinator *comb* as follows. In the type-preserving instance, sequential composition via *seq* is appropriate: the term as returned by the first type-preserving application is passed to the second. In the type-unifying instance, both strategies are evaluated via *pass*, and then, the two resulting values are combined with the binary operation of a monoid. Note that the *Monoid* class is also appropriate to harmonise the parameterisation of the one-step traversal combinators *allTP* and *allTU* as required for overloading. That is, the additional parameters for *allTU* are assumed to correspond to the monoid operations.

In Figure 9, the overloaded combinators for traversal (cf. *one* and *all*) and combination of strategies (cf. *comb*) are defined. For completeness, we also overload *idTP* and *constTU* in a way that an always succeeding strategy is defined (cf. *skip*). The overloaded combinators are placed in two Haskell type classes *GenericMonadPlus* and *GenericMonoid* in order to deal with the different class constraints for the overloaded combinators.

The mother of traversal In Figure 10, we define traversal schemes in three groups. The highly parameterised function *traverse* captures a rich class of traversal schemes as illustrated by the given instantiations. Some of the type-preserving instantiations of *traverse* were identified in [28]. In [15,14,16], some type-unifying

The mother of traversal

$$
\begin{array}{lcl}
traverse\ op\ t\ f & = & f\ `op`\ t\ (traverse\ op\ t\ f) \\
allrec\ op & = & traverse\ op\ all \\
onerec\ op & = & traverse\ op\ one \\
totaltd & = & allrec\ comb \\
totalbu & = & allrec\ (flip\ comb) \\
oncetd & = & onerec\ choice \\
oncebu & = & onerec\ (flip\ choice) \\
stoptd & = & allrec\ choice \\
stopbu & = & allrec\ (flip\ choice)
\end{array}
$$

Traversal with propagation of an environment

$$
\begin{array}{lcl}
propagate\ op\ t\ f\ u\ e & = & (f\ e)\ `op`\ (u\ e\ `pass`\ (\lambda e' \to t\ (propagate\ op\ t\ f\ u\ e'))) \\
totalpe & = & propagate\ comb\ all \\
oncepe & = & propagate\ choice\ one \\
stoppe & = & propagate\ choice\ all
\end{array}
$$

Path schemes

$$
\begin{array}{lcl}
f\ `beloweq`\ p & = & oncetd\ (p\ `pass`\ \lambda() \to oncetd\ f) \\
f\ `below`\ p & = & (one\ f)\ `beloweq`\ p \\
f\ `aboveeq`\ p & = & oncebu\ (oncetd\ p\ `pass`\ \lambda() \to f) \\
f\ `above`\ p & = & f\ `aboveeq`\ (one\ p) \\
belowlist\ f & = & foldl\ below\ f \circ reverse \\
abovelist\ f & = & above\ f \circ foldr\ above\ (constTU\ ()) \\
(ps\ `prepost`\ ps')\ f & = & (f\ `abovelist`\ ps')\ `belowlist`\ ps
\end{array}
$$

Figure 10. Traversal schemes

instantiations were added. In the definition of *traverse*, we separate out the distinguishing parameters of such more specific traversal schemes. The argument *op* is the function combinator to compose the result of node processing and recursive descent. The argument t defines how to descend into children. Finally, f is the node-processing function. Here is the normative type for the mother of traversal:

$$
\begin{array}{lll}
traverse :: Monad\ m & & \\
\quad \Rightarrow (MG\ \kappa\ m \to MG\ \kappa\ m \to MG\ \kappa\ m) & & \text{-- Composition combinator} \\
\quad \to (MG\ \kappa\ m \to MG\ \kappa\ m) & & \text{-- Descent combinator} \\
\quad \to MG\ \kappa\ m & & \text{-- Node-processing function} \\
\quad \to MG\ \kappa\ m & &
\end{array}
$$

We call this a *normative* type because the inferable type is much more general, i.e., $traverse :: (a \to b \to c) \to (c \to b) \to a \to c$. Not even the normative type captures all our intentions, e.g., that t actually leads to descent into children. It is a topic for future work to capture such constraints in suitable types.

The scheme *traverse* is illustrated by several instances. The first two instances *allrec* and *onerec* refine *traverse* to opt either for descent with *all* or *one*, respectively. The postfix "*...td*" or "*...bu*" in some of the names stands for top-down or bottom-up, respectively. The prefixes "*total...*", "*once...*", or "*stop...*" hint on the style of traversal control. All these more concrete schemes are still overloaded as for the choice of *TP* vs. *TU*. To give an example, the scheme *totaltd* models exhaustive traversal of a term where all nodes are processed in top-down manner. Its definition instantiates the argument *op* for the composition of the processed node and the recursive result by *comb*, and the argument t for descent by *all* (via *allrec*).

Traversal with propagation The second group in Figure 10 illustrates a class of traversal schemes which go beyond simple node processing. In addition to traversal, *p*ropagation of an *e*nvironment (abbreviated by the postfix "...*pe*") is performed. The scheme *propagate* takes the same arguments *op*, t, and f as *traverse*. In addition, the scheme carries a generic function argument u to update the environment before descent into the children, and an argument e for the initial environment. Here is the normative type for *propagate*:

$$
\begin{array}{lll}
\mathit{propagate} :: \mathit{Monad}\ m & & \\
\quad \Rightarrow (\mathit{MG}\ \kappa\ m \rightarrow \mathit{MG}\ \kappa\ m \rightarrow \mathit{MG}\ \kappa\ m) & & \text{-- Composition combinator} \\
\quad \rightarrow (\mathit{MG}\ \kappa\ m \rightarrow \mathit{MG}\ \kappa\ m) & & \text{-- Descent combinator} \\
\quad \rightarrow (e \rightarrow \mathit{MG}\ \kappa\ m) & & \text{-- Node-processing strategy} \\
\quad \rightarrow (e \rightarrow \mathit{TU}\ e\ m) & & \text{-- Environment update} \\
\quad \rightarrow e & & \text{-- Initial environment} \\
\quad \rightarrow \mathit{MG}\ \kappa\ m & &
\end{array}
$$

In giving instances of *propagate*, we restrict ourselves to top-down traversal because bottom-up traversal seems to be less obvious when combined with propagation. In is interesting to notice that the scheme *propagate* provides an alternative to using the environment monad [30] in a strategic program.

Path schemes The last group in Figure 10 deals with traversal constrained by the path leading to a node, or starting in a node. For this reason, we do not just use a node-processing strategy f, but we also use predicates p for constraints. The scheme *beloweq* processes a node (not necessarily strictly) below a node for which a predicate holds; dually for *aboveeq*. The *below* and *above* variants enforce the node to be processed and the node constrained by the predicate to be different. We use the traversal schemes *oncetd* and *oncebu* that were derived from *traverse* to organise the path schemes. The types of the path schemes are as follows:

$$
\begin{array}{lll}
\mathit{beloweq}, \mathit{below}, \mathit{aboveeq}, \mathit{above} :: (\mathit{MonadPlus}\ m, \mathit{GenericMonadPlus}\ (\mathit{MG}\ \kappa\ m)) & & \\
\qquad \Rightarrow \mathit{MG}\ \kappa\ m & & \text{-- Node-processing strategy} \\
\qquad \rightarrow \mathit{TU}\ ()\ m & & \text{-- Predicate} \\
\qquad \rightarrow \mathit{MG}\ \kappa\ m & &
\end{array}
$$

These types point out that predicates are encoded as type-unifying functions of type $TU\ ()\ m$ where m is constrained by the *MonadPlus* class. That is, the success and failure behaviour models satisfaction of a predicate. The final three schemes in the group elaborate the more basic path schemes to deal with *lists* of predicates. While the schemes *belowlist* and *abovelist* constrain the prefix or postfix of a node, respectively, the last scheme *prepost* combines these two to process a node on a path with a certain prefix and postfix surrounding the processed node.

The beat of the drum As it was already indicated earlier, the path scheme *belowlist* provides the solution for the introductory traversal problem (recall the illustration in Figure 1). So we think of a path that involves three nodes n_1, n_2, and n_3 in the given order. n_1 and n_2 are recognised via predicates p_1 and p_2 while n_3 is processed via a node processor f. The following functional strategy is appropriate:

$$f\ `\mathit{belowlist}`\ [p_1, p_2]\ \equiv\ (f\ `\mathit{below}`\ p_2)\ `\mathit{below}`\ p_1$$

Note: (i) Node processors and predicates are parameters in our approach. (ii) Traversal behaviour is selected and composed via simple combinators. (iii) Functional strategies are typed. We leave an XSLT encoding of *belowlist* and its subsequent instantiation as an exercise to the reader.

Appraisal

Contribution The four movements of this symphony combine certain bits of parametric polymorphism, type case, polytypism, and overloading to define typed and highly parameterised traversal schemes. We should point out that we rely on *higher-order* and *typed* functional programming to be able to make this contribution. Higher-orderness is crucial because schemes like *traverse* or *propagate* take nullary, unary or binary arguments to assemble an actual traversal from them. A scheme like *belowlist* even folds over a list of argument strategies. Types are indispensable to deal with such highly parameterised entities because inappropriate instantiation is otherwise too easy. Also, type-safe customisation of generic traversal schemes is impossible without types.

Research context and perspective This work is based on earlier work on functional strategies [16] by Joost Visser and the present author. Functional strategies make functional programming fit for generic traversal. The key idea to define traversal schemes in terms of one-step traversal combinators carries over from the seminal work on term rewriting strategies [28]. The present paper reaches a new level of genericity when compared to previous work on strategic programming [28,15,14,16] due to the identified principles for folding over children, and for overloading different type schemes of traversal. A fold over children was first proposed in [14] in a term-rewriting setting but there it was less potent due to the limitations of the underlying type system. Based on experience with modelling functional strategies, we derive a candidate language extension for Haskell in a forthcoming paper [17]. This extension is complementary compared to other proposals for generic programming extensions for Haskell [9,10].

Acknowledgement A concert performance was given at a Generic Haskell Meeting, at Utrecht University, 27 March 2002. The following people contributed beats to this polymorphic symphony: Jan Kort, Andres Löh, Joost Visser, and Stephanie Weirich. I am grateful for the remarks and suggestions by the anonymous WRS'02 referees. I devote my first symphony to Berenike and Bernadette.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, Apr. 1991.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic Typing in Polymorphic Languages. In *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 92–103, San Francisco, June 1992. Association for Computing Machinery.

[3] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[4] M. Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, July 2001.

[5] J. Cockett and D. Spencer. Strong categorical datatypes I. In R. Seely, editor, *Proceedings Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, volume 13 of *Canadian Mathematical Society Conf. Proceedings*, pages 141–169. American Mathematical Society, Providence, RI, 1992.

[6] E. Dolstra and E. Visser. First-class rules and generic traversal. Technical report, Universiteit Utrecht, UU-CS-2001-38, 2001. Technical report.

[7] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[8] R. Harper and G. Morrisett. Compiling polymorphism using intentional type analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 130–141. ACM, Jan. 1995.

[9] R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proc. of the 3rd Haskell Workshop, Paris, France*, Sept. 1999. Technical report, Universiteit Utrecht, UU-CS-1999-28.

[10] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of ENTCS. Elsevier Science, Aug. 2001.

[11] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, Berlin, 1996.

[12] M. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer-Verlag, 1995.

[13] M. Jones. First-class polymorphism with type inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 15–17 Jan. 1997.

[14] R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies, May 2002. 85 pages; To appear in the Journal of Logic and Algebraic Programming; First published as CWI Technical Report in August 2001.

[15] R. Lämmel and J. Visser. Type-safe Functional Strategies. In *Draft proc. of SFP'00, St Andrews*, July 2000.

[16] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C. Ramakrishnan, editors, *Proc. of PADL'02*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[17] R. Lämmel and J. Visser. Yet Another Generic Haskell. In preparation, 2002.

[18] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[19] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[20] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[21] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[22] M. Odersky and K. Läufer. Putting type annotations to work. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*, pages 54–67. ACM, Jan. 1996.

[23] S. Peyton Jones and J. Hughes (eds.). Haskell 98: A non-strict, purely functional language. `http://www.haskell.org/onlinereport/`, 1999.

[24] J. Reynolds. Towards a Theory of Type Structures. In *Programming Symposium (Colloque sur la Programmation, Paris)*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[25] J. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Proc. of 9th IFIP World Computer Congress, Information Processing '83, Paris, France, 19–23 Sept 1983*, pages 513–523. North-Holland, Amsterdam, 1983.

[26] M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 289–302. ACM Press, 1998.

[27] S. Thompson. Higher-order + Polymorphic = Reusable. Unpublishled; `http://www.cs.ukc.ac.uk/pubs/1997/224`, May 1997.

[28] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[29] P. Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, 1989.

[30] P. Wadler. The essence of functional programming. In ACM, editor, *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 1–14. ACM Press, 1992.

[31] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In A.-S. ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 60–76, Austin, TX, USA, Jan. 1989. ACM Press.

[32] S. Weirich. Higher-order intensional type analysis. In D. L. Métayer, editor, *Programming Languages and Systems: 11th European Symposium on Programming, ESOP 2002, Grenoble, France*, number 2305 in LNCS, pages 98–114, 2002.

[33] J. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. of 9th Ann. IEEE Symp. on Logic in Computer Science, LICS'94, Paris, France,* pages 176–185. IEEE Computer Society Press, 1994.

A The *Term* class

In Haskell, the combinators for strategy update and for folding over constructor applications need to be implemented per datatype. To this end, we use the *Term* class. In the sequel, we will briefly describe one possible encoding scheme for the *Term* class. This scheme is supported by the generative tool which comes with this paper (see the paper's web-site).

To start with, here is the declaration of the *Term* class:

$$\begin{array}{l}\mathbf{class}\ \mathit{Term}\ \alpha\ \mathbf{where}\\ \quad \mathit{hfoldr}' :: \mathit{HFoldrAlg}\ \kappa \to \alpha \to \kappa\ \alpha\\ \quad \mathit{adhoc}' :: \mathit{Term}\ a' \Rightarrow \mathit{Generic}\ \kappa \to \mathit{Parametric}\ \alpha\ \kappa \to \mathit{Parametric}\ a'\ \kappa\end{array}$$

We use primed members $adhoc'$ and $hfoldr'$ with (in part) implicitly quantified types. This is necessary because we need to be able to access the corresponding type parameters as class parameters for overloading. We can easily derive the ultimate combinators *adhoc* and *hfoldr* with their first-class polymorphic result types:

$$\begin{array}{l}\mathit{adhoc}\ \mathit{poly}\ \mathit{mono} = G\ (\mathit{adhoc}'\ \mathit{poly}\ \mathit{mono})\\ \mathit{hfoldr}\ \mathit{alg} = G\ (\mathit{hfoldr}'\ \mathit{alg})\end{array}$$

The encoding scheme for $hfoldr'$ is very simple because we only need to provide one equation per constructor for heterogeneous, right-associative fold as discussed in the paper. That is, given a constructor with n arguments, we need a pattern-match case as follows:

$$\mathit{hfoldr}'\ (A\ f\ z)\ (C\ x_n\ \cdots\ x_2\ x_1)\ =\ f\ x_1\ (f\ x_2\ (\cdots\ (f\ x_n\ (z\ C))\ \cdots))$$

The encoding scheme for $adhoc'$ is more involved. Let us assume that we deal with finitely many types, be it k. Recall that we need to be able to compare the domain α of the monomorphic ingredient with the type α' of the eventual input term. To this end, we encode a kind of decision matrix to check for $\alpha = \alpha'$. Such a decision matrix has obviously the size $k * k$. One dimension is taken care of by the mere list of instances for the *Term* class. Looking at the type for $adhoc'$, one can see that the α for the type of the monomorphic update coincides with the class parameter. Hence, the question is how we encode the other dimension, namely the α' for the eventual input term. The trick is here to use a helper member per type to *record* the specific type α from the update while this member is still overloaded in α'. These k helper members in the k instances apply the update *mono* in case $\alpha = \alpha'$, and they resort to the generic default *poly* otherwise. Note that we are faced with a closed-world assumption because the actual declaration of the *Term* class depends on the k types to be covered due to the required helper members. A different scheme without a CWA is discussed in [16]. It makes disciplined use of dynamic typing.

We illustrate the scheme for the sample system from the paper (see Figure 6). The system involves three types, namely the defined types *SortA* and *SortB*, and the basic datatype *Int*.

We need the additional members in *Term* in order to be able to record types for *adhoc*:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Term}\ \alpha\ \textbf{where} \\
\quad \ldots \\
\quad \mathit{int} :: \mathit{Generic}\ \kappa \rightarrow \mathit{Parametric}\ \mathit{Int}\ \kappa \rightarrow \mathit{Parametric}\ \alpha\ \kappa \\
\quad \mathit{sorta} :: \mathit{Generic}\ \kappa \rightarrow \mathit{Parametric}\ \mathit{SortA}\ \kappa \rightarrow \mathit{Parametric}\ \alpha\ \kappa \\
\quad \mathit{sortb} :: \mathit{Generic}\ \kappa \rightarrow \mathit{Parametric}\ \mathit{SortB}\ \kappa \rightarrow \mathit{Parametric}\ \alpha\ \kappa
\end{array}
$$

We can even pre-fill the decision matrix by assuming that we will resort to the generic default most of the time because there is only one case out of k per instance (say, per term type) where we have to apply the monomorphic update. Hence, the following generic member definitions are placed in the *Term* class once and for all:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Term}\ a\ \textbf{where} \\
\quad \ldots \\
\quad \mathit{int}\ \mathit{poly}\ \_ = \mathit{unG}\ \mathit{poly} \\
\quad \mathit{sorta}\ \mathit{poly}\ \_ = \mathit{unG}\ \mathit{poly} \\
\quad \mathit{sortb}\ \mathit{poly}\ \_ = \mathit{unG}\ \mathit{poly}
\end{array}
$$

Here is the instance of *Term* for *Int*:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Term}\ \mathit{Int}\ \textbf{where}} \\
\quad \mathit{adhoc}' & = & \mathit{int} \\
\quad \mathit{int}\ \_\ \mathit{mono} & = & \mathit{mono} \\
\quad \mathit{hfoldr}'\ (A\ \_\ z)\ i & = & z\ i
\end{array}
$$

As for folding, this instance is trivial because an integer cannot be further destructed. Hence, the member hfoldr' is defined in a way to apply the ingredient z for an empty constructor application to the given integer. As for strategy update, the member adhoc' records the type of the update via the overloaded member int. If the eventual input value is of type *Int*, then overloading resolves to the definition of int in the instance for *Int*. This definition encodes coincidence of $\alpha = \alpha' = \mathit{Int}$, and hence, the update *mono* is safely applied. Otherwise, an int from some other instance will reply, and it will resort to the generic default due to the generic definition of the helper members in the *Term* class.

The instances for *SortA* and *SortB* illustrate folding:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Term}\ \mathit{SortA}\ \textbf{where}} \\
\quad \mathit{adhoc}' & = & \mathit{sorta} \\
\quad \mathit{sorta}\ \_\ \mathit{mono} & = & \mathit{mono} \\
\quad \mathit{hfoldr}'\ (A\ f\ z)\ (\mathit{SortA1}\ b) & = & f\ b\ (z\ \mathit{SortA1}) \\
\quad \mathit{hfoldr}'\ (A\ \_\ z)\ \mathit{SortA2} & = & z\ \mathit{SortA2} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Term}\ \mathit{SortB}\ \textbf{where}} \\
\quad \mathit{adhoc}' & = & \mathit{sortb} \\
\quad \mathit{sortb}\ \_\ \mathit{mono} & = & \mathit{mono} \\
\quad \mathit{hfoldr}'\ (A\ f\ z)\ (\mathit{SortB}\ i\ a) & = & f\ a\ (f\ i\ (z\ \mathit{SortB}))
\end{array}
$$

In the case of *SortA*, there are two constructors *SortA1* and *SortA2*. Consequently, the folding scheme needs to be instantiated for both possible forms of constructor applications. In the case of *SortB*. we see how the folding scheme is instantiated for a constructor *SortB* with two components.

[End Of Symphony]

Strategies for Source-to-Source Constant Propagation

Karina Olmos [1,2]

Institute of Information and Computing Sciences
Universiteit Utrecht
Utrecht, The Nederlands

Eelco Visser [3]

Institute of Information and Computing Sciences
Universiteit Utrecht
Utrecht, The Nederlands

Abstract

Data-flow optimizations are usually implemented on low-level intermediate representations. This is not appropriate for source-to-source optimizations, which reconstruct a source level program after transformation. In this paper we show how constant propagation, a well known data-flow optimization problem, can be implemented on abstract syntax trees in Stratego, a rewriting system extended with programmable rewriting strategies for the control over the application of rules and dynamic rewrite rules for the propagation of information.

1 Introduction

Optimizing compilers for imperative languages apply data-flow optimizations to improve the performance of programs [14,2]. Data-flow optimizations such as constant propagation, copy propagation, and dead code elimination transform or eliminate statements or expressions based on data flow information that is propagated along the control-flow paths of the program. For example, in constant propagation the assignment of a constant value to a variable is propagated to occurrences of the variable, which can then be replaced by the constant value.

Data-flow optimizations in compilers are usually performed on an intermediate representation (IR) in which control is expressed by means of labels and

[1] Supported by Philips Research

[2] Email: `karina@cs.uu.nl`

[3] Email: `visser@cs.uu.nl`

jumps, and memory access is expressed in terms of registers and memory stores and fetches [16,14,22], or in terms of stack operations. For the implementation of optimizations, the flat list of instructions is turned into a control-flow graph, which guides the propagation of information. Implementing optimizations on a low-level intermediate representation has the advantage that the implementation is reusable for all source languages that are translated to the IR.

On the other hand, the translation to IR leads to the elimination of information which was available at source level. In compilation this leads to overhead, e.g., control-flow analysis is required to recover control-flow structures such as conditional branches and loops, which were available in the original source program. In source-to-source transformation the loss of information makes it impossible to recreate source programs close to the original.

Source-to-source transformation systems manipulate programs at source level and produce transformed programs in the same (high-level) language. Applications include instrumentation, aspect weaving, software renovation, and domain-specific optimization. For example, the CodeBoost transformation system [4] transforms C++ programs with the purpose of eliminating the overhead caused by an abstract programming style, and applying optimizations based on knowledge of the application domain, which the C++ compiler does not have.

To (re)construct a source level program after transformation, source-to-source transformations are best applied to a representation that is as close to the original source as possible, i.e., abstract syntax trees. Since transformations on control-flow graphs are not directly applicable to abstract syntax trees, source-to-source optimizers cannot reuse optimizations on control-flow graphs.

Term rewriting suggests itself as a natural paradigm for transformations on (abstract syntax) trees. While optimizations such as algebraic simplification and constant folding can indeed be expressed elegantly using rewrite rules, data-flow optimizations such as constant propagation and copy propagation cannot be expressed directly using rewriting. First of all rewrite rules are context-free, i.e., can only access local information, while in data-flow optimizations the propagation of context information is required. Furthermore, standard rewriting strategies apply rewrite rules exhaustively throughout the tree. In data-flow optimizations, the control-flow paths of the program should be followed.

In this paper we show how data-flow optimizations on abstract syntax trees can be implemented in Stratego [19], a rewriting system extended with programmable rewriting strategies [20] for the control over the application of rules and dynamic rewrite rules [18] for the propagation of information.

To illustrate the ideas we develop the specification of constant propagation for Tiger [2]—an imperative language with nested functions. In Section 2 we present the abstract syntax of Tiger expressions and define constant folding rules and a generic strategy for simplification of Tiger expressions. In Section 3 we add assignment and sequential composition to the language. Constant propagation for such straight-line programs can be defined using dynamic rules, which take care of substituting constants for the variables to which they have been assigned. The strategy ensures that the rules are applied in the right order. In Section 4 we add structured control-flow constructs such as conditionals and loops. Propagation of constant assignments forks at such control flow statements and needs to be combined at the meeting points. Appropriate operations support saving, restoring, and computing 'meet' operations on sets of dynamic rules.

The building blocks developed for the specification of constant propagation can be considered as a framework for the implementation of data-flow optimizations. We have used these building blocks to implement copy propagation, common sub-expression elimination and dead code elimination. In Section 5 we discuss this generalization, compare the approach to related work, and mention future work.

We assume that the reader is familiar with the basic concepts of term rewriting. We will explain the Stratego constructs that are used, but not in depth. For an introduction to the basic concepts of Stratego we refer the reader to [20,18].

2 Constant Folding

Constant folding is the simplification of operator expressions with known constant values as operands, i.e., reducing expressions of the form `c1 op c2`, for some operator `op` with constant arguments `c1` and `c2`. For example, the expression `3 + (6 * (5 - 2)) / 2` can be simplified at compile time to `12`, by applying the laws of arithmetic. Constant folding is the goal of constant propagation; if a variable can be replaced by its constant value, run-time computations may be replaced with compile-time computations.

Before we study constant propagation, we consider the specification of constant folding on expressions. Figure 1 presents the signature of abstract syntax trees representing Tiger expressions. Expressions consist of arithmetic and relational operations on integer values, boolean values and strings. Boolean values in Tiger are represented with integer values, `false` is represented with the integer zero and `true` with any non zero integer value.

2.1 Constant Folding Rules

The simplification of operators with constant operands can be expressed directly using rewrite rules. Figure 2 presents several constant folding rules for arithmetic and Boolean operators. The rules are labeled such that they can be used in a strategy.

As part of the construction of a new term on the right-hand side of a rule, a strategy `s` can be applied to a subterm `t` using the notation `<s>t`. In the constant folding rules this is used to compute integer operations. For

```
module Tiger-Expressions
signature
 constructors
  Int    : IntConst -> Exp
  String : String -> Exp
  Var    : Id -> Var
  BinOp  : BinOp * Exp * Exp -> Exp
  RelOp  : RelOp * Exp * Exp -> Exp
  Call   : Var * List (Exp) -> Exp
  NilExp : Exp
module Operators
signature
 constructors
  PLUS : BinOp  MINUS : BinOp  MUL : BinOp  DIV  : BinOp
  AND  : BinOp  OR    : BinOp  EQ  : RelOp  NE   : RelOp
  LT   : RelOp  LE    : RelOp  GE  : RelOp  GT   : RelOp
```

Fig. 1. Abstract syntax tree of Tiger expressions

```
EvalBinOp: BinOp(PLUS, Int(i), Int(j)) -> Int(<add>(i,j))
EvalBinOp: BinOp(PLUS, Int(0), e) -> e
EvalBinOp: BinOp(PLUS, e, Int(0)) -> e
EvalBinOp: BinOp(MINUS,Int(i),Int(j)) -> Int(<subt>(i,j))
EvalBinOp: BinOp(MINUS, Var(x), Var(x)) -> Int(0)
EvalBinOp: BinOp(MUL, Int(i), Int(j)) -> Int(<mul>(i,j))
EvalBinOp: BinOp(MUL, Var(x), Int(1)) -> Var(x)
EvalBinOp: BinOp(MUL, Int(1), Var(x)) -> Var(x)
EvalBinOp: BinOp(MUL, Int(0), e) -> Int(0)
EvalBinOp: BinOp(MUL, e, Int(0)) -> Int(0)
EvalBinOp: BinOp(DIV, Int(i), Int(j)) -> Int(<div>(i,j))
           where <not(0)> j
EvalRelOp: RelOp(EQ, Int(a), Int(b)) -> Int(<eval-rel(eq)>(a,b))
EvalRelOp: RelOp(NE, Int(a), Int(b)) ->
           Int(<eval-rel(not(eq))>(a,b))
EvalRelOp: RelOp(GT, Int(a), Int(b)) -> Int(<eval-rel(gt)>(a,b))
EvalRelOp: RelOp(LT, Int(a), Int(b)) -> Int(<eval-rel(lt)>(a,b))
EvalRelOp: RelOp(GE, Int(a), Int(b)) -> Int(<eval-rel(geq)>(a,b))
EvalRelOp: RelOp(LE, Int(a), Int(b)) -> Int(<eval-rel(leq)>(a,b))
```

Fig. 2. Constant folding rules for Tiger expressions

example, `<add>(i,j)` denotes a call to the primitive `add` for addition of the integer values `i` and `j`. The strategy `eval-rel(s)` applies the parameter strategy `s` and turns success into the integer value `1` and failure into `0`.

2.2 Constant Folding Strategy

In Stratego rewrite rules are not applied automatically. Instead, the specification should determine which rules are applied where and in what order by defining a rewriting strategy. For the application of the constant folding rules a single, bottom-up pass over the expression tree suffices to reduce all possible constant operator applications. This can be specified with the generic term traversal `bottomup`, as follows:

```
constant-folding = bottomup(try(fold))
```

The `fold` strategy used in the definition of `constant-folding` is defined as the choice between the `EvalBinOp` and `EvalRelOp` rules from Figure 2:

```
fold = EvalBinOp + EvalRelOp
```

The operator `s1 + s2` denotes the choice between the strategies `s1` and `s2`. Finally, the generic traversal strategy `bottomup` traverses a tree by first applying itself recursively to all direct subterms of a node using `all` and then applying the parameter strategy `s`:

```
bottomup(s) = all(bottomup(s)); s
```

Note that the `bottomup(s)` strategy is a *one pass* traversal.

3 Constant propagation

Constant propagation is defined as the process to discover values that are constant on all possible executions of a program and to propagate these values through the program. This is a common data-flow problem and the results of this specification can be generalized towards different analyses.

For simplicity, we first consider a reduced version of Tiger which consists of expressions, assignment, let and sequential composition. Figure 3 defines this language. When reduced to this syntax, constant propagation comes down to propagating constant assignments through a sequence of assignments. It is important that constants are only substituted for variables in program points where the constant assignment is valid. For example, in the transformation

```
a := 3
b := a + 2
a := y
b := a + b
```

⇒

```
a := 3
b := 5
a := y
b := a + 5
```

the first occurrence of `a` in a right-hand side is replaced by `3`, but the second is not because of the intervening definition of `a`.

3.1 Dynamic Propagation Rules

As the example above illustrates, constant propagation works by replacing a variable with the constant value which is assigned to it. Since this information is not available at the place where a variable is used, the association should be established at the place where the variable is defined, i.e., the assignment, and

```
module Tiger-Statements
signature
  constructors
    Assign : LValue * Exp -> Exp
    Let    : List (Dec) * List (Exp) -> Exp
    VarDec : Id * Option (TypeId) * Exp -> Dec
    Seq    : List (Exp) -> Exp
```

Fig. 3. Reduced abstract syntax of Tiger.

be available at usage sites. Dynamic rewrite rules [18] are designed for exactly this purpose. The following `assign-cp` strategy recognizes an assignment statement and generates a dynamic rule `PropConst` which rewrites the variable `Var(x)` to the expression `e` assigned to it:

```
assign-cp =
  ?Assign(Var(x), e);
  where(<is-value> e
        < rules(PropConst: Var(x) -> e)
        + rules(PropConst: Var(x) -> Undefined))
```

The match construct `?t`, matches the term pattern `t`, just like the left-hand side of a rewrite rule. The `s1 < s2 + s3` construct is a guarded deterministic choice operator. If the guard strategy `s1` succeeds, `s2` is applied to the result of `s1`, else `s3` is applied to the original term. In this case the `PropConst` rule is only generated when `e` is a value, i.e., a constant. In the other case the dynamic rule is *undefined*.

The `rules(L : t1 -> t2)` construct generates a new (dynamic) rewrite rule with label `L`, which rewrites a term matching `t1` to `t2`, just like an ordinary rule `L : t1 -> t2`. The difference is that variables in `t1` and `t2` that are bound in the context are inherited from that context. Thus, in the case above, the (meta)variables `x` and `e` are instantiated when the rule is generated, and not when it is applied. The `assign-cp` strategy can be applied several times for the same variable while a program is being traversed. Every time that this strategy is applied a new rule is generated overriding a previous rule for the same variable. However, rules for different variables are not overriden.

Undefining the `PropConst` rule is necessary in order to kill a previously generated rule *for the same variable*. For instance, in the previous example the rule `PropConst: Var("a") -> Int(3)` is undefined by the assignment of `Var("y")` to `Var("a")`. At this point, the rule is overriden with `PropConst: Var("a") -> Undefined`. The application of an undefined rule always fails. Thus, attempting to apply this rule fails and the term `Var("a")` remains unchanged. Rules for different varialbes are not undefined.

3.2 Propagation Strategy

The propagation strategy `const-prop1` for straight-line code is defined in Figure 4. The strategy is almost the same as the `constant-folding` stra-

```
module Tiger-ConstProp
strategies
 const-prop1 = bottomupS(try(fold1 + assign-cp), control-flow1)
 fold1 = PropConst + fold
 control-flow1(cp) = Assign(id, cp)
```

Fig. 4. Constant propagation for straight line code.

tegy of the previous section. That is, during a bottom-up traversal over the program, constant folding rules are applied. The combined constant folding rules `fold` of the previous section are extended with the dynamically generated `PropConst` rules and with the `assign-cp` strategy for generating these rules. Thus, if a variable is encountered during constant folding, the dynamic rule `PropConst` is applied to discover its constant value and replace it with that value. If the variable is not associated with a constant, the rule will not succeed, and the variable remains in the program.

However, the strategy cannot be a complete `bottomup` traversal. Consider, for example, the sequence `a := 3; a := a + 2`. Figure 5 illustrates the visit order through the abstract syntax tree for this sequence using the generic `bottomup(s)` traversal. The numbers on the edges indicate at which point the parameter strategy `s` is applied to the node the edge points to. Clearly the first assignment is visited before the variables in the second one, thus correctly propagating the constant `3` to the variable `a` in the second assignment. However, the variable `a` in the *left-hand side* of the second assignment would also be replaced. To avoid this the constant propagation traversal should not visit the left-hand sides of assignments. This is achieved with the `bottomupS` strategy, a variant of `bottomup`, which is defined as:

```
bottomupS(s, skip: a * (a -> a) -> a) =
  (skip(bottomupS(s, skip))  <+ all(bottomupS(s, skip))) ; s
```

The `skip` parameter is a strategy which provides an alternative for the generic traversal with `all`. In Figure 4 the `skip` strategy is instantiated with the `control-flow1` strategy, which only applies the constant propagation strategy to the right-hand side of assignments. This is achieved using the strategy expression `Assign(id, cp)`, a *congruence* strategy which specifies the traversal to the subterms of a specific constructor. In general, a congruence strategy `c(s1,...,sn)` matches with a term `c(t1,...,tn)` and constructs the term `c(<s1>t1,...,<sn>tn)`, applying the argument strategies to the corresponding argument terms. Thus, `Assign(id,cp)` applies the identity strategy `id` to the left-hand side of an assignment and constant propagation `cp` to the right-hand side. Thus, achieving a traversal that is faithful to the control flow of the programming language. In Figure 5, the corresponding visit order is the one where visits 1 and 4 are skipped.

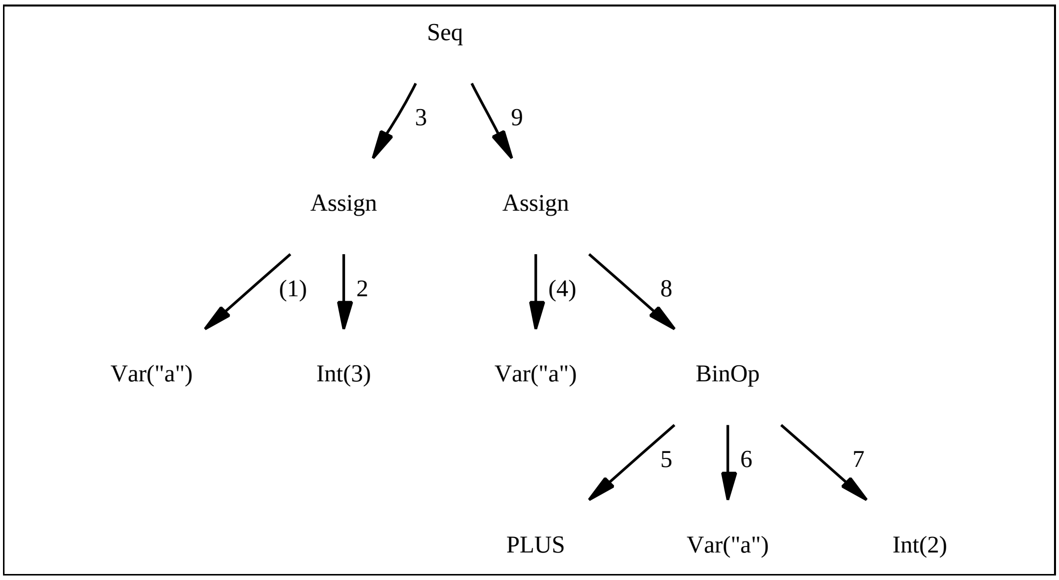


Fig. 5. Visit order for `bottomup(s)`

4 Propagation and Control-Flow

So far we have considered constant propagation in straight line code. Real programs use conditionals and loops which fork or iterate the data flow. Control-flow statements in Tiger consist of the `if-then-else` and `if-then` conditional statements, and the `for` and `while` iterative statements. Figure 6 gives the abstract syntax of these constructs.

The propagation of constant values on control-flow statements forks and merges according to different execution paths. The propagation of constant values has to follow the different paths imposed by the data flow of the program. For instance, consider the following transformation:

```
b := 4;
d := 2;
if b > x then
   (a := 6 * d;
    b := 45)
  else
   (b := 6;
    a := b + b);
c := a + d + b
```

⇒

```
b := 4;
d := 2;
if 4 > x then
   (a := 12;
    b := 45)
   else
   (b := 6;
    a := 12);
c := 14 + b
```

where the variables `a, b, d` are of interest: `b` and `d` contain constant values before the occurrence of the `if` statement and these values are propagated to the branches of the `if`. After the `if`, `a` and `d` contain constant values, `a` contains a constant value, because it is defined with the same constant value in both branches. The variable `d` is not defined in the if statement. On the other hand, `b` is assigned a different value in each branch, hence it does not contain a constant value after the `if`, even though it does contain a constant value *locally* in the branches.

```
module Tiger-Statements
signature
 constructors
  IfThen : Exp * Exp -> Exp
  If     : Exp * Exp * Exp -> Exp
  While  : Exp * Exp -> Exp
  For    : Var * Exp * Exp * Exp -> Exp
```

Fig. 6. Control-flow statements of Tiger.

4.1 Unreachable Code Elimination

Control-flow statements fork the execution paths of a program. At execution time some paths are not reachable. To avoid considering unreachable code, the constant folding rules are extended to eliminate branches from control-flow statements when conditions can be evaluated statically. Consider the `If` statement with three argument expressions, a `condition`, a `then`, and an `else` expressions. Constant propagation first evaluates the `condition` (as the execution of the program will do). If this evaluation results in a constant value, it determines which branch will be executed (Tiger uses 0 for false and another integer value for true). Unreachable code elimination can be expressed by means of rewrite rules as is depicted in Figure 7. The application of `ElimIf` and `ElimIfThen` rules select the executable branch and the `If` expression is replaced by the surviving branch. The search for constants will continue in the selected branch. If a loop will not be executed it can be removed from the program.

4.2 Distributing and Merging Propagation Rules

The specification of constant propagation is based on dynamic rules to propagate constant values. Dynamic propagation rules must follow the data flow of a program. To achieve this behavior, propagation rules are passed to the execution paths and merged at the point where execution paths are joined. Figure 8 depicts the data flow of an if statement. A structured if-then-else statement forks and merges two execution paths. At the point where the if-then-else statement forks, dynamic propagation rules are passed to the `then` and `else` branches. At the merge point, two different versions of dynamic

```
ElimIf    : If(Int(0), e1, e2) -> e2
ElimIf    : If(Int(i), e1, e2) -> e1 where <not(eq)> (i, 0)
ElimIfThen: IfThen(Int(0),e1) -> Seq([])
ElimIfThen: IfThen(Int(i),e1) -> e1
ElimWhile : While(Int(0),e) -> Seq([])
ElimFor   : For(v,Int(i),Int(j),e) -> Seq([])
            where <eval-rel(lt)> (j,i) => 1
```

Fig. 7. Rules to eliminate unreachable code

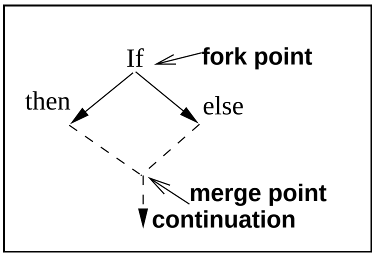


Fig. 8. Data flow of an structured If statement

Input Term	*Transformation Rules*		*Transf. Term*
	Active rules	*Sets of rules*	
`b := 4;`	`PC: Var(b) -> 4` *(1)*		`b := 4;`
`d := 2;`	`PC: Var(d) -> 2` *(2)*		`d := 2;`
`if b>x then`	*saving active rules*	`dr-in ={PC: Var(b) -> 4,` `PC: Var(d) -> 2}`	`if 4>x then`
`(a := 6 * d;` `b := 45 )`	`PC: Var(a) -> 12` *(3)* `PC: Var(b) -> 45` *(4)*		`(a := 12;` `b := 45)`

`else`	*saving active rules*
	
	`PC: Var(b) -> 4` *(5)*
	`PC: Var(d) -> 2` *(6)*
		`dr-then ={PC: Var(d) -> 2,`
		`PC: Var(a) -> 12,`
		`PC: Var(b) -> 45}`
		restoring dr-in rules
			`else`

`(b := 6;` `a := b + b);`	`PC: Var(b) -> 6` *(7)* `PC: Var(a) -> 12` *(8)*	*overrides rule (5)*	`(b := 6;` `a := 12);`

	$\bigcap$ `dr-then =`
	`{PC: Var(d) -> 2,`
	`PC: Var(a) -> 12}`
		intersection with active rules (6,7,8)	

`c:= a + d + b`	`PC: Var(c) ->` *Undefined*		`c := 14 + b`

Fig. 9. Example of how dynamic rules are used to propagate constants

propagation rules are combined.

Propagation rules carry the information required to propagate constants. These rules have to traverse the program in a way that emulates the data flow of a program. In order to manipulate active propagation rules, operations such as save, restore and intersect(merge) are used. These operations allow to modify active propagation rules.

Figure 9 illustrates how distributing and merging of propagation rules is accomplished. The example is decorated with two extra columns to illustrate the result of operations on dynamic rules. For space reasons, the name of the rule `PropConst` is depicted as `PC`. The first and last column represent the input and the transformed term. The intermediate columns are introduced to show how dynamic rules are used to propagate constant values. The second column shows the state of the active rules. The third column shows the result of an operation on dynamic rules which uses extra storage. Each row in the table shows an inspected node, the result of an operation on dynamic rules, and the transformed term. The first and the second row of the table generate a propagation rule respectively. The if statement evaluates the condition and saves active propagation rules in `dr-in` to be passed to its branches. The `then` branch is traversed and the resulting state of propagation rules is stored in `dr-then` set. To traverse the `else` branch with the state of incoming propagation rules to the if statement, rules in `dr-in` set are restored. At the merge point of the if statement an intersect operation is specified to obtain the active propagation rules valid after the if statement.

The specification of constant propagation for an if statement is defined by the strategy:

```
if-cp(cp) =
 If(id, id, id)
   ; where(save-PropConst => dr-in)
   ; If(id, cp, id)
   ; where(save-PropConst => dr-then; <restore-PropConst> dr-in)
   ; If(id, id, cp)
   ; where(<isect-PropConst> dr-then)
```

This strategy uses the operations `save-PropConst`, `restore-PropConst`, and `isect-PropConst` to manipulate the dynamic rule. The `where(s)` strategy applies a strategy `s` to a term without modifying the term.

In order to distribute incoming active rules to the branches of an if, the active set of propagation rules is saved. Constant propagation continues traversing the `then` branch, the result of this inspection is saved in the `dr-then` set of dynamic rules. To proceed in the `else` branch, a restore operation is defined to activate the `dr-in` set. With this state, constant propagation continues in the `else` branch. To merge two different versions of propagation rules from both branches, an intersect operation is specified. The intersection is specified with the current state of dynamic rules (active rules considering the else branch) and the `dr-then` set. A propagation rule is maintained in the active set if it is defined in both sets and rewrites to the same constant value.

4.2.1 Generalization

The data flow schema for `if` statements above was specific for constant propagation. It turns out that the schema can be reused for other optimizations by using different save, restore, and merge operations. This generalization can be formalized by abstracting the schema over these operations, as follows:

```
if(s, save, restore, merge) =
 If(id, id, id)
   ; where(save => dr-in)
   ; If(id, s, id)
   ; where(save=> dr-then; <restore> dr-in)
   ; If(id, id, s)
   ; where(<merge> dr-then)
```

An example of instantiation of this schema is `if(cp, save-PropConst, restore-PropConst, isect-PropConst)` for constant propagation. Different

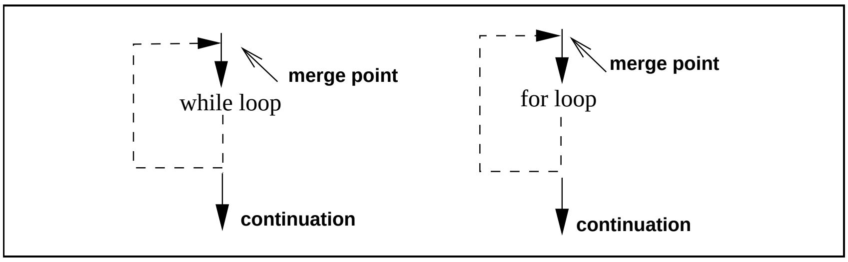


Fig. 10. Merging points of control flow statements.

analyses and optimizations can reuse this building block by providing different strategies to operate on dynamic rules. In the specification of the other control-flow constructs we take this general approach into account.

4.3 Loop statements

Loop statements execute a number of times (including zero) the body of a loop statement. The execution flow of a loop has a reentering path as is depicted in Figure 10. Therefore, loop statements forks the data flow and iterate the body of the loop.

Structured loop statements execute the iteration conditions as the first and last execution. As a motivating example for constant propagation on loops; consider the following code:

```
(a := 5;
 b := 4;
 c := 8;
 d := 5;
 while((c < 100)|x)
 do(b := a;
    e := a + b;
    a := e * d + c;
    c := c + e;
    e := e + c;
    a := b);
 f := a + b + d + c + e)
```

⇒

```
(a := 5;
 b := 4;
 c := 8;
 d := 5;
 while((c < 100)|x)
 do(b := 5;
    e := 10;
    a := (50 + c);
    c := (c + 10);
    e := (10 + c);
    a := 5);
 f := (15 + c + e))
```

In this example the variables `a,b,c` and `d` contain constant values before the while statement is inspected. This example illustrates the following situations:

- `a` contains a constant value, regardless of the execution of the while statement.
- `b` is defined in the body of the while, if the while is executed at least one iteration, b contains the value of `a`, i.e `5`, and it can be propagated outside the while statement.
- `c` does not contain a constant value because it is iteratively defined in the while body.
- `d` contains a constant value since it is not defined in the while.

```
while-p(s,save,restore,merge)=
  While(id, id)
  ; where(save => dr)
  ; where(While(id, s))
  ; where(While(id, s))
  ; where(<merge> dr)
  ; While(s, s)
  ; where(<merge> dr)
```

```
for-p(s,save, restore,merge)=
  For(id, id, id, id)
  ; where(save => dr)
  ; where(For(id, id, id, s))
  ; where(For(id, id, id, s))
  ; where(<merge> dr)
  ; For(id, id, id, s)
  ; where(<merge> dr)
```

Fig. 11. Data flow schemas for loops

- `e` contains a constant value from its first definition until its second definition inside the while loop. After the while `e` does not contain a constant value.

At the point when the assignment to `f` is reached, the variables `a`, `b` and `d` contain constant values.

The specification of constant propagation has to cover all these cases. To discover the value of `a` as containing a constant value, we inspect the while body twice, in order to take into account propagation rules discovered in the first inspection. This analysis considers the reentering path of the loop. After the unfolded inspection of the while, the merge operation selects valid propagation rules considering possible execution paths. With the outcome of valid propagation rules the transformation is performed. A last merge operation is required to prevent propagation rules coming from the while execution being active outside the scope of the while loop. The `for` statement is defined in an analogous way.

The definition to propagate constant values on loop statement is shown in Figure 11.

4.3.1 Non Pessimistic Constant Propagation

The constant propagation specification so far is still pessimistic, even though we avoid to consider unreachable code. Loop statements that are not executed are removed from the program when the condition can be evaluated. If the analysis cannot determine if a loop will execute or not, it considers both situations. That is the reason for the second occurrence of `merge` for loop statements. This specification cannot determine the value of `b` as constant in the last example.

A less pessimistic approach is to keep propagation rules after a loop statement when the analysis guarantees at least one execution of the body of a loop. The `is-safe` strategy checks this condition and avoids to perform the second `merge` of propagation rules. Thus, constant propagation rules of a loop statement are active after the loop occurrence in the program. The strategies in Figure 12 consider these situations.

```
is-safe =
 ?While(Int(i),e);
 where(<not(eq)> (i,0);
       <not(oncetd(Break))> e)

while(s,save,restore,meet) =
 While(id,id)
 ; where(save => dr)
 ; (where(While(s,id);is-safe)
    ; where(While(id,s))
    ; where(While(id,s))
    ; where(<meet> dr)
    ; While(s,s)
 <+ where(While(id,s))
    ; where(While(id,s))
    ; where(<meet> dr)
    ; While(s,s)
    ; where(<meet> dr)
 )
```

```
is-safe =
  ?For(_,Int(i),Int(t),_);
  where( <geq> (t, i))


for(s,save,restore,merge) =
  For(id,id,id,id)
  ; where(save => dr)
  ; where(For(id,id,id,s))
  ; where(For(id,id,id,s))
  ; where(<merge> dr)
  ; For(id,id,id,s)
  ; ( is-safe
     <+ where(<merge> dr) )
```

Fig. 12. Non-pessimistic loop data flow schemas

```
const-prop2 =
 bottomupS(
  try(fold2(const-prop2) <+
  eval-conditionals2(const-prop2))
  , control-flow2
 )

control-flow2(s) =
    Assign(id, s)
 + IfThen(s, id)
 + If(s, id, id)
 + For(id, s, s, id)
 + While(id, id)
 + Let([try(VarDec(id,id,s))]
        , id)

fold2(s) =
   fold1
 + ElimIf; s
 + ElimFor
 + ElimIfThen; s
 + While(s,id); ElimWhile
```

```
eval-conditionals2(s)=
  eval-conditionals(s,
        assign-cp,
        declVar,
        save-PropConst,
        restore-PropConst,
        isect-PropConst)

eval-conditionals(s,
    assign, declare,
    save, restore, meet)=
  assign
+ ifthen(s,save,restore,meet)
+ if(s,save,restore,meet)
+ for(s,save,restore,meet)
+ while(s,save,restore,meet)
+ let-exp(s,declare)

let-exp(s, declare) =
  Let([try(declare)],list(s))
```

Fig. 13. Constant propagation for control-flow statements

4.4 Strategies for Control-flow Statements

Now we have defined all ingredients for putting together a full blown constant propagation strategy. Figure 13 combines the ingredients into a specification of non-pessimistic intra-procedural constant propagation. The `const-prop2` strategy is defined in terms of the `bottomupS` strategy and includes strategies for control-flow statements. The non-pessimistic constant propagation uses the evaluation rules `ElimIf, ElimIfThen, ElimWhile`, and `ElimFor`, to simplify the execution flow of expressions when possible. The strategy `fold2` is parameterized with the `cp` strategy to apply constant propagation to the resulting paths after the simplification of statements.

When constant folding reduction is not applicable we consider different execution paths as explained before. The `eval-conditionals2` strategy is defined in terms of `assign-cp`, and the data flow schemas for conditionals and loops. The `control-flow1` strategy is extended with terms that need control when the reduction strategy is applied. The strategy `control-flow2` defines how to traverse control-flow statements to simulate the execution of the program.

5 Discussion

We have shown how to specify intra-procedural constant propagation using strategic dynamic term rewriting. The techniques used in the paper such as dynamic scoped rules and context-sensitive term traversal permitted us to write a specification which is elegant and concise.

5.1 Generalization and Future Work

As presented here, the constant propagation optimization is intra-procedural and does not take function calls into account. We have specified an extension that analyzes function definitions and at function calls only undefines propagation rules for global variables killed in the function body. Generalization to inter-procedural constant propagation [7,5] needs further investigation, but seems straightforward.

Other source-to-source optimizers can be built using the same building blocks by providing different propagation rules and corresponding rule operations. We have already specified copy propagation, common sub-expression elimination and dead code elimination using this approach. We are confident that the techniques can be used for the specification of many other machine independent optimizations such as code motion [11], although this requires further investigation.

The current building blocks can be easily extended to include control-flow constructs that are not present in Tiger such as case and switch. Also extension to semi-unstructured control-flow using breaks can be dealt with. The framework can be made reusable with little effort across different languages.

Different dynamic rules can be created to propagate or to provide information for different aspects of a program. At the moment we are interested in investigating the possibility of combining several optimizations while a program is being inspected in the style of [13].

Constant propagation or in general information propagation is been used in partial evaluators [15], supercompilers [10], interpreters [8], as a few examples. In [8] dynamic rules are used to propagate information of the evaluation state of expressions. The present work can be extended and combined with other techniques such as program specialization and generalization in order to construct such systems.

5.2 Previous Work

This paper is part of an ongoing investigation into the specification of different kinds of program transformations using rewriting strategies and dynamic rules. In [20] programmable rewriting strategies, in particular for generic traversal, were introduced and applied to simplification of functional programs. In [18] dynamic rules are introduced and applied to bound variable renaming, function inlining, and dead function elimination. In [6] dynamic rules are used to implement dynamic programming for instruction selection, i.e., associating the lowest cost reduction with a term. The Tiger optimization presented in this paper is part of the Tiger-in-Stratego [4] project, which is aimed at exploring program transformation techniques for compilation.

The contribution of this paper is the formalization of merge operations for dynamic rules and the traversal strategy that emulates the control flow of a program. These mechanisms allow us to write the building blocks that are reusable for other optimizations that require a *forward* analysis. These building blocks can be parameterised for different optimizations, providing different information through dynamic rules for each aspect that is relevant to inspect. A *backward* traversal of the program was implemented using similar constructions with dead code elimination as an instance.

5.3 Related Work

Path logic programming [9] is a related specification technique for data-flow optimizations. The applicability conditions of a transformation rule are expressed by means of regular expressions over the paths through a program. For example, a constant assignment can be propagated to a usage site if there are no intervening definitions of the variable on any path between definition and use. Instead of transforming abstract syntax trees, control-flow graphs are used as basis for transformations. The specification of optimizations is more declarative than the one presented in this paper. On the other hand, our approach allows for the simultaneous propagation of all constants or other

[4] `http://www.stratego-language.org/Hpc/TigerInStratego`

properties throughout the program, whereas in the path logic programming approach, the entire graph needs to be reanalyzed after each transformation.

Expressing transformations by rewriting tends to be clear and intuitive. There are three major factors to consider for a rewrite system. Expressiveness of transformations, representation of control-flow paths, and data availability to test applicability conditions to perform rewriting. Rewriting is split into two different areas: term rewriting and graph rewriting [3]. Term rewriting expresses transformations in a clear and elegant way, but it is more difficult to express applicability conditions, when data not available in the inspected term is needed, term rewriting systems suffers from this issue. In contrast, graph rewriting provides mechanisms to represent the control flow of a program, the applicability conditions of rewriting is also easier to represent, although the rewriting itself is more difficult to represent and to provide mechanisms for these operations [12]. Providing information to term rewriting by means of dynamic rules and a strategy to emulate the data flow of programs leads to a high level specification for constant propagation which is executable.

There are many algorithms for constant propagation, most of them use a graph representation such as control-flow graph with the *static single assignment* (SSA) property [21], or even *gated single assignment* property [17]. Special ϕ assignments are introduced to provide the representation of a program with unique variable definitions. The most used algorithm for constant propagation is the *Sparse conditional constant propagation* (SCC) [21,17,14,2]. SCC is an optimistic algorithm which avoids to consider unreachable code and continues the propagation of constant values in the branches of conditionals statements. These algorithm properties are also present in our specification.

Another approach uses *system dependence graph* (SDG) to represent the data-flow and control-flow of a program [5]. This representation focuses on data dependencies of a program, which is represented by means of graphs. This system is provided with a data-flow interpreter to propagate constant values to the dependent nodes. Assignment nodes provide the hints to propagate constants in the graph. At each node the SDG graph is updated following transformation rules. The SDG approach is more convenient for source-to-source optimizers than algorithms based on control-flow graphs with SSA property, since they do not lower the representation language. Program reconstruction preserves the structure of the program, although building the system dependence graph is required in order to perform the transformation.

In all publications we encountered (with [9] as exception) algorithms for constant propagation are expressed using pseudocode or using formal English to describe properties of a system. Our specification is consice and executable without the need for def-use chains [1] or similar analysis information. All information required is present in the rewriting rules and the mechanisms to operate on them suffices for our specification.

6 Conclusion

In this paper we have described the application of programmable rewriting strategies to the specification of a data-flow optimization. Programable strategies allow us to emulate the data flow of a program, even though using abstract syntax trees to represent a program. The parameterization of the traversal strategies with the rules to apply makes it possible to define different optimizations by providing a different set of rules and even enables us to combine different optimizations.

The use of dynamic rules for the propagation of data-flow information (instead of environments) makes it possible to separate traversal from propagation. The extension of dynamic rules with operations to save, restore, and intersect sets of rules makes it possible to model data-flow through multiple paths.

Together, these mechanisms support the combination of analysis and transformation while the abstract syntax tree is traversed.

References

[1] Aho, A., R. Sethi and J. Ullman, "Compilers: Principles, Techniques, and Tools," Addison-Wesley, 1986.

[2] Appel, A., "Modern compiler implementation in ML," Cambridge University Press, 1998.

[3] Assmann, U., *How To Uniformly Specify Program Analysis and Transformation*, in: P. Fritzson, editor, *Compiler Construction (CC)* (1996).

[4] Bagge, O., M. Haveraaen and E. Visser, *Codeboost: A framework for the transformation of c++ programs*, Technical report, Institute of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands (2001).

[5] Binkley, D., *Interprocedural constant propagation using dependence graphs and a data-flow model*, in: *Computational Complexity*, 1994, pp. 374–388.

[6] Bravenboer, M. and E. Visser, *Rewriting strategies for instruction selection*, in: S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, Lecture Notes in Computer Science (2002), (To appear).

[7] Carini, P. R. and M. Hind, *Flow-sensitive interprocedural constant propagation*, in: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California*, SIGPLAN Notices **30/6**, 1995, pp. 23–31.

[8] Dolstra, E. and E. Visser, *Building interpreters with rewriting strategies*, in: M. van den Brand and R. Laemmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, Electronic Notes in Theoretical Computer Science **65.3** (2002).

[9] Drape, S., O. de Moor and G. Sittampalam, *Transforming the .net intermediate language using path logic programming*, Submitted to : Conference on Principles and Practice of Declarative Programming (2002).

[10] Glück, R. and M. H. Sørensen, *A roadmap to metacomputation by supercompilation*, in: O. Danvy, R. Glück and P. Thiemann, editors, *Partial Evaluation*, Lecture Notes in Computer Science **1110** (1996), pp. 137–160.

[11] Knoop, J., O. Ruthing and B. Steffen, *Partial dead code elimination*, in: *SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 147–158.

[12] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, LNCS **2027** (2001), pp. 52–68.

[13] Lerner, S., D. Grove and C. Chambers, *Combining dataflow analyses and transformations*, in: *SIGPLAN Symposium on Principles of Programming Languages (POPL 2002)*, 2002.

[14] Muchnick, S., "Advanced compiler design and implementation," Morgan Kaufmann Publishers, 1997.

[15] Sakharov, A., *Specialization of imperative programs through analysis of relational expressions*, in: O. Danvy, R. Glück and P. Thiemann, editors, *Partial Evaluation*, Lecture Notes in Computer Science **1110** (1996), pp. 430–445.

[16] Standford Compiler Group, *The SUIF library,1.0 edition* (1994).

[17] Stoltz, E., M. Wolfe and M. Gerlek, *Constant propagation: A fresh demand-driven look*, in: *In ACM Symposium on Applied Computing, Phoenix, Arizona, March 1994. ACM SIGAPP.*, 1994.

[18] Visser, E., *Scoped dynamic rewrite rules*, in: M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, Electronic Notes in Theoretical Computer Science **59/4** (2001).

[19] Visser, E., *Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5*, in: A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science **2051** (2001), pp. 357–361.

[20] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, ACM SIGPLAN Notices **34** (1999), pp. 13–26, proceedings of the International Conference on Functional Programming (ICFP'98).

[21] Wegman, M. and F. Zadeck, *Constant propagation with conditional branches*, ACM Transactions on Programming Languages and Systems (TOPLAS) **13** (1991), pp. 181–210.

[22] Wilson, R., R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," ACM SIGPLAN Notices, 29(12) (1996).

Optimal Strategies in Higher-Order Rewriting

Vincent van Oostrom [1]

Faculteit der Wijsbegeerte, Universiteit Utrecht
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

[1] Email: Vincent.vanOostrom@phil.uu.nl

An Abstract Böhm-normalization

John Glauert [1]

School of Information Systems, UEA
Norwich NR4 7TJ, UK

Zurab Khasidashvili [2]

Department of CS, Bar-Ilan University
Ramat-Gan 52900, Israel

Abstract

We study *normalization by neededness* with respect to 'infinite results', such as Böhm-trees, in an abstract framework of *Stable Deterministic Residual Structures.* We formalize the concept of 'infinite results' for finite terms as suitable sets of infinite reductions, and prove an abstract infinitary normalization theorem with respect to such sets. We also give a sufficient and necessary condition for existence of *minimal* normalizing reductions.

1 Introduction

A normalizable term in a rewriting system may have an infinite reduction, so it is important to have a *normalizing* strategy which enables one to construct reductions to normal form. For example, it is well known that the leftmost-outermost strategy is normalizing in the λ-calculus [11]. For Orthogonal TRSs, Huet and Lévy [17] found a general normalizing strategy that always contracts a *needed* redex, one with a residual that has to be contracted in any reduction to normal form.

This fundamental work on neededness has been extended in several directions. Barendregt et al. [5], Maranget [34], Nöcker [38] and Middeldorp [37] studied neededness w.r.t. head-normal forms, weak head-normal forms, constructor head-normal forms, and root-stable forms, respectively. Kennaway et al. [20] studied a needed strategy for infinitary orthogonal TRSs. A different approach to normalization is developed in Kennaway [18] and Antoy and Middeldorp [2]. Khasidashvili [22,23] introduced a refinement of the needed

[1] Email: `jrwg@sys.uea.ac.uk`
[2] Email: `khasidz@cs.biu.ac.il`

strategy which is based on a concept of descendant which refines that of residual. Melliès [36] and van Oostrom [40] extended the needed and essential strategies, respectively, to some non-orthogonal systems.

In [13], the present authors addressed the question of normalization relative to a desired set of final terms, considering the properties that a set $\mathcal{S}$ of terms must possess in order for the neededness theory of Huet and Lévy still to make sense. This work is done in the context of orthogonal *Expression Reduction Systems* (ERSs), a formalism for rewriting which subsumes Term Rewriting and the λ-calculus. Natural conditions were imposed on $\mathcal{S}$, called *stability*, that are sufficient for the following *Relative Normalization* (RN) theorem to hold: each $\mathcal{S}$-normalizable term not in $\mathcal{S}$ (that is, not in $\mathcal{S}$-*normal form*) has at least one $\mathcal{S}$-needed redex, and repeated contraction of such redexes always leads to an $\mathcal{S}$-normal form. It is shown also that if a stable $\mathcal{S}$ is *regular*, i.e., if $\mathcal{S}$-unneeded redexes cannot duplicate $\mathcal{S}$-needed ones, then the $\mathcal{S}$-needed strategy is hypernormalizing as well, and *minimal* (w.r.t. the Lévy-embedding relation) $\mathcal{S}$-normalizing reductions exist.

In [12], we further generalized the theory by abstracting from the concrete structure of terms. We studied relative normalization in Deterministic Residual Structures (DRSs), which are abstract reduction systems with an axiomatized residual relation. Despite their highly abstract nature, a counterpart of the *stability* property of Berry [7] and Winskel [43] enabled us to prove the RN theorem for all *regular* stable sets $\mathcal{S}$. (We actually proved the Relative Hypernormalization theorem.) We showed that without this stability axiom the theorem fails. The proof method employed is similar to that in [22,23], and is based on the fact that $\mathcal{S}$-needed steps in a reduction can be pushed before $\mathcal{S}$-unneeded steps without affecting the number of $\mathcal{S}$-needed steps. In [25], we studied *discrete normalization* in SDRSs, which is normalization relative to particular (finite or infinite) reductions.

Here we extend the concepts of stability and regularity from sets of terms (which represent 'finite' results, or values, such as head normal forms) to sets of reductions, to represent the concepts of 'infinite' values, such as Böhm trees. Throughout the paper, we restrict ourselves to finite terms – terms can contain only a finite number of redexes. Other authors have considered terms with an infinite number of redexes and transfinite reductions [20,21], but such extensions are outside the scope of the present work.

For regular stable sets $\mathcal{S}$ of finite or infinite results and an $\mathcal{S}$-normalizable term t, in an SDRS, we show that all $\mathcal{S}$-needed fair reductions starting from t are $\mathcal{S}$-normalizing, and if t has a finite $\mathcal{S}$-normalizing reduction, then any $\mathcal{S}$-needed reduction starting from t eventually normalizes t in a finite number of steps. This result unifies the Relative Hypernormalization theorem with the Discrete Normalization theory. We also show that for $\mathcal{S}$-minimal reductions to exist, for regular stable sets of finite results $\mathcal{S}$, every $\mathcal{S}$-normalizing term must possess an $\mathcal{S}$-needed $\mathcal{S}$-erased redex (where a redex is $\mathcal{S}$-erased if it does not have a residual under $\mathcal{S}$-normalizing reductions).

The paper is organized as follows. In the next section, we recall SDRSs and present some fundamental lemmas concerning (mutually) *external* reductions. In section 3, we give a construction of the meet operation with respect to Lević's ordering on reductions. In Section 4, we introduce the concept of stability for sets of reductions, and give examples. In Section 5, we prove the RN theorem for regular stable sets of reductions in SDRSs, and present the minimality results. Conclusions appear in Section 6.

2 Deterministic Residual Structures

In this section we recall *Deterministic Residual Structures* (DRSs) which are *Abstract Reduction Systems* (ARSs) satisfying certain properties concerning *residuals*. Residuals of redexes were first introduced and studied by Church and Rosser in the λI-calculus [10]. The study of abstract systems with residuals started from Hindley [15].

The definition and some results about ARSs can be found e.g., in [29,16]. Our definition is slightly different.

Definition 2.1 An ARS is a triple $A = (Ter, Red, \to)$ where Ter is a set of *terms*, ranged over by t, s, o, e; Red is a set of *redexes* (or *redex occurrences*), ranged over by u, v, w; and $\to$: $Red \to (Ter \times Ter)$ is a function such that for any $t \in Ter$ there is only a finite set of $u \in Red$ such that $\to(u) = (t, s)$, written $t \xrightarrow{u} s$. This set will be known as the redexes of term t, where $u \in t$ denotes that u is a member of the redexes of t and $U \subseteq t$ denotes that U is a subset of the redexes. Note that $\to$ is a *total* function, so one can identify u with the triple $t \xrightarrow{u} s$.

A *reduction* is a sequence $t \xrightarrow{u_1} t_2 \xrightarrow{u_2} \ldots$. Reductions are denoted by P, Q, N. We write $P : t \twoheadrightarrow s$ or $t \overset{P}{\twoheadrightarrow} s$ if P denotes a reduction (sequence) from t to s. $Q : t \twoheadrightarrow$ may be finite as well as infinite. Reductions are *co-initial* if they share the same initial term. $P + Q$ denotes the concatenation of P and Q. u also denotes the reduction that contracts u. Further, for any reduction Q, $\lceil Q \rceil^k$ will denote the initial part of Q of length k, provided $k \leq |Q|$ (the length of Q), and $\lceil Q \rceil_k$ will denote the tail of Q starting from the $(k+1)$th step; thus $Q = \lceil Q \rceil^k + \lceil Q \rceil_k$. Finally, we write $P \leq Q$ if P is an initial part of Q.

Definition 2.2 (Deterministic Residual Structure) A *Deterministic Residual Structure* (DRS) is a pair $\mathcal{R} = (A, /)$, where A is an ARS and $/$ is a *residual* relation on redexes relating redexes in the source and target term of every reduction $t \xrightarrow{u} s \in A$, such that for $v \in t$, the set v/u of *residuals of* v *under* u is a set of redexes of s; a redex in s may be a residual of only one redex in t under u, and $u/u = \emptyset$. If v has more than one u-residual, then u *duplicates* v. If $v/u = \emptyset$, then u *erases* v, and if moreover $v \neq u$, then u *discards* v. A redex of s which is not a residual of any $v \in t$ under u is said to be u-*new* or *created* by u. The set v/P of residuals of v under any finite reduction P is defined by transitivity. $v/P = \{v\}$ if $|P| = 0$.

A *development* of $U \subseteq t$ is a reduction $P : t \twoheadrightarrow$ that only contracts residuals of redexes from U; it is *complete* if it is finite and $U/P = \cup_{u\in U} u/P = \emptyset$. Development of $\emptyset$ is identified with the empty reduction. The residual relation satisfies the following two axioms:

[FD] (*Finite Developments* [14]) All developments are terminating; all complete developments of $U \subseteq t$ end at the same term; and residuals of a redex $v \in t$ under all complete developments of U are the same. Below U will also denote a complete development of $U \subseteq t$.

[weak acyclicity] ([42]) Let $u, v \in t$, $u \neq v$, and $u/v = \emptyset$. Then $v/u \neq \emptyset$.[3]

We call a DRS $\mathcal{R}$ *stable* (SDRS) if the following axiom is satisfied:

[stability] If $u, v \in t$ are different redexes, $t \xrightarrow{u} e$, $t \xrightarrow{v} s$, and u creates a redex $w \in e$, then the redexes in $w/(v/u)$ are not u/v-residuals of redexes of s, i.e., they are created along u/v.

$$\begin{tikzcd}
t \arrow[r, "u"] \arrow[d, "v"'] & e \arrow[r, "w"] \arrow[d, two heads, "v/u"] & \cdot \\
s \arrow[r, two heads, "u/v"'] & o \arrow[r, two heads, "w/(v/u)"'] & \cdot
\end{tikzcd}$$

DRSs are closely related to *Determinate Concurrent Transition Systems* (DCTS) of Stark [42], and to *Abstract Reduction Systems* (ARSs) of Gonthier et al. [14]. The main difference from DCTSs is that Stark requires a non-duplicating residual relation. Unlike ARSs of [14], we do not have a nesting relation on redexes and the corresponding axioms, and the stability axiom is modified accordingly. Instead, we study properties of conflict-free transition and reduction systems based on the duplication and erasure effect of executed transitions on the others, and develop a theory that is applicable to systems with nested transitions too. Other related abstract reduction systems are studied in [35,39,41].

One can verify that all orthogonal (first or fully extended higher-order, see e.g., [41]) term rewriting systems, such as the λ-calculus, form DRSs. These systems are stable, and can be shown so just using an appropriate notion of descendant which assigns the contractum to the contracted redex – labelling is not necessary. Further, orthogonal Term Graph Rewriting Systems [19] are DRSs but they do not satisfy all the nesting axioms of [14].

The properties of the residual relation in orthogonal systems are all standard [17,31,8,32,9,42,35], and we only review quickly the main constructions used in this paper. In a DRS $\mathcal{R}$, *Lévy-equivalence* or *permutation-equivalence* is defined as the smallest relation on co-initial finite reductions satisfying: $U+V/U \approx_L V+U/V$ and $Q \approx_L Q' \;\Rightarrow\; P+Q+N \approx_L P+Q'+N$, where U and V are sets of redexes in the same term. The residual relation on redexes is extended to all co-initial reductions as follows: $(P_1+P_2)/Q = P_1/Q+P_2/(Q/P_1)$ and $Q/(P_1+P_2) = (Q/P_1)/P_2$. The *Lévy-embedding* relation on co-initial fi-

[3] This axiom is called [acyclicity] in [12], and is axiom (4) in [42].

nite reductions, $\trianglelefteq_L$, is defined by: $P \trianglelefteq_L Q$ iff $P/Q = \emptyset$. One can show that $P \approx_L Q$ iff $P \trianglelefteq_L Q$ and $Q \trianglelefteq_L P$ and that $P \trianglelefteq_L Q$ iff $Q \approx_L P + N$ for some N. Intuitively, $P \trianglelefteq_L Q$ expresses that Q does more work than P, and Q/P is the part of Q that remains after P has been done. Further, one can prove the following *strong Church-Rosser* theorem for DRSs: For all co-initial finite reductions P and Q in a DRS, $P \sqcup Q \approx_L Q \sqcup P$, where $P \sqcup Q$ abbreviates $P + Q/P$. And finally, we will also need the following *Cube Lemma*: for any finite co-initial reductions P, Q and N, $N/(P \sqcup Q) = N/(Q \sqcup P)$.

Lévy-equivalence extends to infinite reductions as follows. First, for any $u \in t \overset{P}{\twoheadrightarrow}$, define $u/P = \emptyset$ if $u/P' = \emptyset$ for some finite $P' \leq P$ (we say u is *erased* in P). Now, for any co-initial P, Q, define $P/Q = \emptyset$, or equivalently, $P \trianglelefteq_L Q$, if for any redex v contracted in P, say $P = P' + v + P''$, $v/(Q/P') = \emptyset$ (see the diagram); and define $P \approx_L Q$ iff $P \trianglelefteq_L Q$ and $Q \trianglelefteq_L P$.

$$\begin{tikzcd}
\cdot \arrow[r, two heads, "P'"] \arrow[d, "Q"'] & \cdot \arrow[r, "v"] \arrow[d, "Q/P'"] & \cdot \arrow[r, two heads, "P''"] & P \\
{} & {} & &
\end{tikzcd}$$

Finally, P/Q is only defined for finite Q, as the reduction whose initial parts are residuals of the initial parts of P under Q.

The [stability] axiom is not used in the above constructions. Intuitively, stability means that a redex cannot arise from two unrelated sources. This property has a natural extension to many-step reductions, where 'unrelated' is formalized by the notion of *external* which captures the concept that two external reductions do not contract redexes that have common residuals (although the contracted redexes may have 'inessential' common ancestors).

Definition 2.3 • Let $u \in U \subseteq t$ and $P : t \twoheadrightarrow o$. We call P *external* to U (resp. u) if P does not contract residuals of redexes in U (resp. residuals of u).

- Let $P : t_0 \overset{P_i}{\twoheadrightarrow} t_i \overset{u_i}{\to} t_{i+1} \twoheadrightarrow t_n$ and $Q : t_0 = s_0 \overset{Q_j}{\twoheadrightarrow} s_j \overset{v_j}{\to} s_{j+1} \twoheadrightarrow s_m$. Let $U_{i,j} = u_i/(Q_j/P_i)$ and $V_{i,j} = v_j/(P_i/Q_j)$ (see diagram). We call P *external* to Q if for any i, j, $U_{i,j} \cap V_{i,j} = \emptyset$.

$$\begin{tikzcd}
t_0 \arrow[r, two heads, "P_i"] \arrow[d, two heads, "Q_j"'] & t_i \arrow[r, "u_i"] \arrow[d, two heads, "Q_j/P_i"'] & t_{i+1} \arrow[d, two heads] \\
s_j \arrow[r, two heads, "P_i/Q_j"] \arrow[d, "v_j"'] & \cdot \arrow[r, two heads, "U_{i,j}"] \arrow[d, two heads, "V_{i,j}"'] & \cdot \arrow[d, two heads] \\
s_{j+1} \arrow[r, two heads] & \cdot \arrow[r, two heads] & \cdot
\end{tikzcd}$$

Obviously, P is external to the set U iff it is external to any development of U, and is external to a redex u iff it is external to the reduction contracting u. Note that a reduction external to one complete development of U need not be external to all developments of U, and in general, externality is not invariant under $\approx_L$. For, consider a TRS $R = \{a \to a',\ f(x) \to b,\ g(x) \to c\}$, a term $t = f(g(a))$, and reductions $P : t \xrightarrow{a} f(g(a')) \xrightarrow{f} b$, $Q : t \xrightarrow{a} f(g(a')) \xrightarrow{g} f(c)$, and $N : t \xrightarrow{g} f(c)$. Then we have $Q \approx_L N$, P is external to N, but not to Q; and P is not external to $U = \{a,\ g(a)\}$.

Recall from [17] that in concrete orthogonal rewrite systems, a redex is said to be *external* if its residuals never appear inside arguments of other redexes. It should be clear from the context to which concept of external we are referring.

Lemma 2.4 *Let $P : t_0 \xrightarrow{v_0} t_1 \xrightarrow{v_1} \ldots \to t_n$ be external to $U = \{u_1, \ldots, u_n\} \subseteq t_0$, and let $Q_0 : t_0 \twoheadrightarrow o_0$. Then $P' = P/Q_0$ is external to the set $U' = U/Q_0$.*

We conclude this section by presenting two fundamental lemmas, which generalize the [weak acyclicity] and [stability] axioms to many-step reductions [12].

Lemma 2.5 (Weak Acyclicity) *Let P, N be co-initial finite reductions in a DRS, and let P be external to N. Then $N \not\approx_L P$.*

Lemma 2.6 (Stability) *Let $P : t \twoheadrightarrow s$ be external to $Q : t \twoheadrightarrow e$, in a stable DRS, and let P create redexes $W \subseteq s$. Then the residuals $W/(Q/P)$ of redexes in W are created by P/Q, and Q/P is external to W.*

$$\begin{array}{ccc} t & \xrightarrow{\quad P \quad} & s \supseteq W \\ {\scriptstyle Q}\Big\downarrow & & \Big\downarrow{\scriptstyle Q/P} \\ e & \xrightarrow[P/Q]{} & o \supseteq W/(Q/P) \end{array}$$

3 Construction of the $\trianglelefteq_L$-meet operation

In this section we give a construction of the $\trianglelefteq_L$-meet operation (to be used in the subsequent sections) and prove its correctness. This implies that $\trianglelefteq_L$ is a complete lattice on Lévy-equivalence classes of reductions. This lattice was first constructed by Berry and Lévy [8], for the case of Recursive Program Schemes (RPSs), as the ideal completion of the upper semi-lattice on Lévy-equivalence classes of finite reductions.

Below, in our construction of $\trianglelefteq_L$-meets, Φ denotes a set of co-initial reductions in a DRS. Further, for example, we write $P \trianglelefteq_L \Phi$ iff $\forall Q \in \Phi.\ P \trianglelefteq_L Q$; $\lceil \Phi \rceil^k$ denotes the set of initial parts, of length k, of reductions in Φ; etc.

Definition 3.1 ($\trianglelefteq_L$-meet) Let Φ be a set of reductions starting from t, in a DRS. Then the $\trianglelefteq_L$*-meet* of reductions in Φ, written $\sqcap_L \Phi$, is defined as follows: Let $U \subseteq t$ be the maximal subset such that $U \trianglelefteq_L \Phi$, and let $t \overset{U}{\twoheadrightarrow} s$ be a complete U-development (or the multi-step contracting U). Then $\sqcap_L \Phi = U + \sqcap_L(\Phi/U)$.

Theorem 3.2 *Let Φ be a set of co-initial reductions in a DRS. Then $\sqcap_L \Phi$ is a (unique up to $\approx_L$) $\trianglelefteq_L$-meet of Φ.*

Proof. *Let $\sqcap_L \Phi = Q : t_0 \overset{U_0}{\twoheadrightarrow} t_1 \overset{U_1}{\twoheadrightarrow} t_2 \twoheadrightarrow \ldots$. It is immediate from the construction that $\sqcap_L \Phi \trianglelefteq_L \Phi$. Thus we need to show that for any $P : t_0 \twoheadrightarrow$ such that $P \trianglelefteq_L \Phi$, $P \trianglelefteq_L Q$, that is, for any $n < |P|$, $\lceil P \rceil^n \trianglelefteq_L Q$. We show this by induction on n. The case $n = 0$ (i.e., $P = \emptyset$) is clear. So let $n = k+1$, and let $\lceil P \rceil^k \trianglelefteq_L Q$. Then $\lceil P \rceil^k \trianglelefteq_L \lceil Q \rceil^{l_k}$ for some l_k. We can assume that $\lceil Q \rceil^{l_k}$ ends at t_m for some m. Thus $\lceil Q \rceil^{l_k} \approx_L \lceil P \rceil^k + Q'_k$ for some finite Q'_k, and $\Phi / \lceil Q \rceil^{l_k}$ consists of Q'_k-residuals of reductions in $\Phi / \lceil P \rceil^k$, up to $\approx_L$. Let v be the $(k+1)$th step of P. Since $P \trianglelefteq_L \Phi$, $v \trianglelefteq_L \Phi / \lceil P \rceil^k$, thus by the Cube Lemma, $(v/Q'_k) \trianglelefteq_L \Phi / \lceil Q \rceil^{l_k}$, and therefore $v/Q'_k \subseteq U_m$ by Definition 3.1. This means that $\lceil P \rceil^n \trianglelefteq_L Q$. Thus $P \trianglelefteq_L Q$, and $\sqcap_L \Phi$ is a $\trianglelefteq_L$-meet of Φ.*

$$
\begin{tikzcd}
 & & {} & {} \\
t_0 \arrow[r, two heads, "\lceil P\rceil^k"] \arrow[d, "N \in \Phi"] & \cdot \arrow[r, two heads, "Q'_k"] \arrow[d, "N/\lceil P\rceil^k"] \arrow[ur, "v"] & t_m \arrow[r, two heads, "U_m"] \arrow[d, "N/\lceil Q\rceil^{l_k}"] \arrow[ur, "v/Q'_k"] & \cdot \\
{} & {} & {} &
\end{tikzcd}
$$

□

4 Stable sets of finite or infinite results

In this section, we will introduce the concepts of stability, regularity and superstability of sets of reductions, to cover usual concepts of infinite results, such as Böhm-trees. From this definition, our earlier concepts of stability and regularity for sets of finite results [12,13] can be obtained as a special case; moreover, [28] shows that with any reduction P one can associate a stable set $\mathcal{S}_P$ of reductions, which will allow one to infer Discrete Normalization and Standardization results [25] from our normalization results in the next section.

Definition 4.1 Given a set $\mathcal{S}$ of reductions in a DRS $\mathcal{R}$ and a term t in $\mathcal{R}$, we call t $\mathcal{S}$-*normalizable* if $\mathcal{S}$ contains a reduction P starting from t.

Definition 4.2 Let $\mathcal{S}$ be a set of reductions in a DRS.

(1) We call $u \in t$ $\mathcal{S}$-*unneeded* if there is a reduction $Q \in \mathcal{S}$ starting from t that is external to u, and call it $\mathcal{S}$-*needed* otherwise. We call a reduction P $\mathcal{S}$-*(un)needed* if it contracts only $\mathcal{S}$-(un)needed redexes.

(2) Let P, Q be co-initial reductions. If P and Q are both finite, then we define $P \trianglelefteq_{\mathcal{S}} Q$ if P/Q is $\mathcal{S}$-unneeded. Otherwise, $P \trianglelefteq_{\mathcal{S}} Q$ if for any finite $P' \leq P$ there is a finite $Q' \leq Q$ such that $P' \trianglelefteq_{\mathcal{S}} Q'$. Further, $P \approx_{\mathcal{S}} Q$ iff $P \trianglelefteq_{\mathcal{S}} Q$ and $Q \trianglelefteq_{\mathcal{S}} P$. We call $\trianglelefteq_{\mathcal{S}}$ and $\approx_{\mathcal{S}}$ respectively $\mathcal{S}$-*embedding* and $\mathcal{S}$-*equivalence*.

It is immediate from the definition that $\trianglelefteq_L \subseteq \trianglelefteq_{\mathcal{S}}$.

Lemma 4.3 *Let P, Q be co-initial reductions in a DRS. Then $P \trianglelefteq_L Q$ implies $P \trianglelefteq_{\mathcal{S}} Q$ and $P \approx_L Q$ implies $P \approx_{\mathcal{S}} Q$.*

Definition 4.4 Let $\mathcal{S}$ be a set of reductions in a DRS.

(1) We call $\mathcal{S}$ *stable* iff:

[**CS**] $\mathcal{S}$ *is suffix-closed*: if $P' \notin \mathcal{S}$, then $P' + P'' \in \mathcal{S}$ implies $P'' \in \mathcal{S}$.

[**CE**] $\mathcal{S}$ *is closed under* $\mathcal{S}$*-embedding*: $P \in \mathcal{S}$ and $P \trianglelefteq_{\mathcal{S}} Q$ implies $Q \in \mathcal{S}$.

[**CN**] $\mathcal{S}$ *is closed under neededness*: every non-empty $P \in \mathcal{S}$ contracts at least one $\mathcal{S}$-needed redex.

(2) Furthermore, we call $\mathcal{S}$ *regular* iff:

[**Reg**] In no term can an $\mathcal{S}$-unneeded redex duplicate an $\mathcal{S}$-needed one.

(3) Finally, we call $\mathcal{S}$ *superstable* iff:

[**Min**] For any $\mathcal{S}$-normalizable term t, $\mathcal{S}$ contains a unique, up to $\approx_L$, $\trianglelefteq_L$-minimal element starting from t. Such reductions will be called $\mathcal{S}$-minimal.

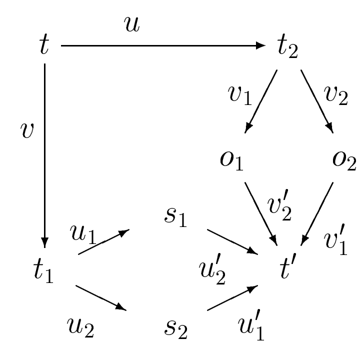
Every set of terms $\mathcal{S}$ is associated with a set $\mathcal{S}_{red}$ of ($\mathcal{S}$-normalizing) reductions starting outside $\mathcal{S}$ and entering it. The induced concepts of *stability*, *regularity*, etc. for $\mathcal{S}$ coincide exactly with earlier concepts in [13].

Examples of stable sets are normal forms [17], head normal forms in the λ-calculus [5], weak head normal forms in the λ-calculus [1], constructor head normal forms for constructor TRSs [38], and root stable forms in orthogonal TRSs [37]. All the above sets are regular. Other examples include weak-head-normal forms (up to garbage-collection, modulo a congruence) in Yoshida's λf-calculus (an environment calculus) [45], the set of *answers* in the *call-by-need* λ-calculus of Ariola et al. [3], and flexible generalized head normal forms in the λ^v_{hd}-calculus of Xi [44]; all are conditional rewrite systems.

Although all normalizing reductions are minimal w.r.t. the set of normal forms, this is not so for other concepts of results. For example, for the set of head-normal form, only the reductions Lévy-equivalent to the leftmost-outermost head-normalizing reductions are minimal. Head-minimal reductions compute the principal head-normal forms [4], from which all other head-normal forms are accessible.

An example of an irregular stable set is given in [12]. The next example exhibits a regular stable set which is not superstable:

Example 4.5 Consider the SDRS given by the following diagram



where $u/v = \{u_1, u_2\}$, $v/u = \{v_1, v_2\}$, $v_1/v_2 = \{v_1'\}$, $v_2/v_1 = \{v_2'\}$, $u_1/u_2 = \{u_1'\}$, $u_2/u_1 = \{u_2'\}$, and let $\mathcal{S} = \{o_2, s_2, t'\}$. Then $\mathcal{S}$ is regular stable but not superstable, as one can easily verify that there is no $\mathcal{S}$-minimal reduction starting from t.

It is easy to check that for example the sets of reductions 'computing' Böhm-trees [4] or Lévy-Longo-trees [30,33] or Berarducci-trees [6] are stable and regular. To make this precise, let us consider the Böhm-tree semantics of the λ-calculus. Recall that the *immediate syntactic value* $\omega(t)$ of a λ-term t is defined by: $\omega(t) = \bot_{\mathcal{B}}$ if t is not a head normal form, and $\omega(t) = \lambda x_1 \ldots x_m.y\omega(t_1)\ldots\omega(t_n)$ if $t = \lambda x_1 \ldots x_m.yt_1 \ldots t_n$, where $\bot_{\mathcal{B}}$ is a special constant which denotes the 'undefined'. Then the *Böhm-tree* or *Böhm-normal-form* $BT(t)$ of t, which is the *value* of t according to *Böhm-semantics*, is defined by $BT(t) = \trianglelefteq_\Omega\text{-} lub\{\omega(t) \mid t \twoheadrightarrow_\beta s\}$, where $\trianglelefteq_\Omega$ is the minimal context-closed ordering on λ-$\bot_{\mathcal{B}}$-terms (i.e., λ-terms possibly containing occurrences of $\bot_{\mathcal{B}}$) generated by $\bot_{\mathcal{B}} \trianglelefteq_\Omega t$ for any λ-$\bot_{\mathcal{B}}$-term t.

Now let us define the *Böhm-approximant computed by* $P : t_0 \rightarrow_\beta t_1 \rightarrow_\beta \ldots$, written $\omega(P)$, by $\omega(P) = \trianglelefteq_\Omega\text{-}\ lub\{\omega(t_i) \mid, i = 0, 1 \ldots\}$. It is well known that $t \twoheadrightarrow_\beta s$ implies $\omega(t) \trianglelefteq_\Omega \omega(s)$, thus in particular for a finite $P : t_0 \twoheadrightarrow_\beta t_k$, $\omega(P) = \omega(t_k)$. Now we can define that $P : t_0 \twoheadrightarrow_\beta$ *computes* $BT(t_0)$ iff $\omega(P) = BT(t_0)$. Let us call a subterm of t an $\bot_{\mathcal{B}}$-subterm if its corresponding occurrence in $\omega(t)$ is $\bot_{\mathcal{B}}$. Recall that the standard way to approximate $BT(t_0)$ sequentially is to construct 'iterated head-reductions' $P : t_0 \overset{P_0}{\twoheadrightarrow}_\beta t_1 \overset{P_1}{\twoheadrightarrow}_\beta \ldots$, where P_i is the head-reduction computing the principal head normal form of one of the $\bot_{\mathcal{B}}$-subterms of t_i having a head normal form. In order to compute $BT(t_0)$, P must be 'fair', i.e., every $\bot_{\mathcal{B}}$ subterm of every t_i having a head normal form must eventually be dealt with in P; we call such reductions P *canonical Böhm-reductions.*

Regularity and stability of the set $\mathcal{S}_{\mathcal{B}}$ of reductions computing Böhm-trees follow easily from the top-down manner of construction of Böhm-trees: [CS] is immediate; [CN] follows from the fact that head redexes of $\bot_{\mathcal{B}}$-subterms of any λ-term t are $\mathcal{S}_{\mathcal{B}}$-needed; and [CE] follows from a simple observation that $Q \in \mathcal{S}_{\mathcal{B}}$ iff $P \trianglelefteq_{\mathcal{S}_{\mathcal{B}}} Q$ for some canonical Böhm-reduction P. The axiom [Reg] follows from the fact that if a β-redex $u \in t$ is nested in an $\mathcal{S}_{\mathcal{B}}$-unneeded β-redex v, then for any canonical Böhm-reduction $P : t \twoheadrightarrow_\beta$, P does not contract residuals of v, and therefore it does not contract residuals of u as any residual of u is nested inside a residual of v. Finally, [Min] follows from the fact that $P \trianglelefteq_L \sqcap_L \mathcal{S}_{\mathcal{B}}^*$ for any canonical Böhm-reduction P, where $\mathcal{S}_{\mathcal{B}}^*$ is the subset of reductions in $\mathcal{S}_{\mathcal{B}}$ that are co-initial with P (note that, since $P \in \mathcal{S}_{\mathcal{B}}$, $P \trianglelefteq_L \sqcap_L \mathcal{S}_{\mathcal{B}}^*$ implies $P \approx_L \sqcap_L \mathcal{S}_{\mathcal{B}}^*$). To prove $P \trianglelefteq_L \sqcap_L \mathcal{S}_{\mathcal{B}}^*$, assume $P : t = t_0 \overset{u_0}{\rightarrow}_\beta t_1 \overset{u_1}{\rightarrow}_\beta \ldots$ and $\sqcap_L \mathcal{S}_{\mathcal{B}} : t = s_0 \overset{U_0}{\twoheadrightarrow}_\beta s_1 \overset{U_1}{\twoheadrightarrow}_\beta \ldots$. Since u_0 is external, it must be contracted by any reduction in $\mathcal{S}_{\mathcal{B}}$, hence $u_0 \in U_0$ and hence $u_0/U_0 = \emptyset$. Further, P/U_0 remains external, and we can prove similarly that the first contracted redex in P/U_0 must belong to U_1, and so on. This means that, $P/\sqcap_L \mathcal{S}_{\mathcal{B}} = \emptyset$, that is, $P \trianglelefteq_L \mathcal{S}_{\mathcal{B}}$.

Note that $N \trianglelefteq_{\mathcal{S}_{\mathcal{B}}} Q$ implies $\omega(N) \trianglelefteq_{\Omega} \omega(Q)$, but $N \triangleleft_{\mathcal{S}_{\mathcal{B}}} Q$ need not imply $\omega(N) \triangleleft_{\Omega} \omega(Q)$. This is because the ordering $\trianglelefteq_{\mathcal{S}_{\mathcal{B}}}$ is more subtle than $\trianglelefteq_{\Omega}$ and can take into account (or observe) parts of computations that occur in the $\perp_{\mathcal{B}}$-subterms, while $\omega(t) = \perp_{\mathcal{B}}$ for any t not in head normal form, irrespectively of 'how far' is t from being a head normal form. For example, let $t = Ix(Ix)$ and $N : t \xrightarrow{I}_{\beta} (Ix)x$; then $\emptyset \triangleleft_{\mathcal{S}_{\mathcal{B}}} N$ but $\omega(\emptyset) = \omega(N) = \perp_{\mathcal{B}}$. Thus $\trianglelefteq_{\mathcal{S}_{\mathcal{B}}}$, defined on reductions rather than on terms, can be seen as a refinement of the classical Böhm topology on trees [4].

Thus stability of sets of reductions is a natural generalization of stability for sets of finite results. It is also consistent with the definition of stability of sets of configurations in prime event structures with erasure in [26,27].

5 Relative Normalization in SDRSs

In this section, we will generalize the Relative Hypernormalization theorem to regular stable sets of finite or infinite reductions, in stable DRSs $\mathcal{R}$. We then show how $\mathcal{S}$-minimal reductions can be constructed in $\mathcal{R}$.

We begin by showing that, for finite or infinite results $\mathcal{S}$, $\mathcal{S}$-unneeded redexes cannot create $\mathcal{S}$-needed ones, and that residuals of $\mathcal{S}$-unneeded redexes remain unneeded. When $\mathcal{S}$ is regular, this enables us to construct an $\mathcal{S}$-needed variant of any $\mathcal{S}$-normalizing reduction, in any SDRS.

Lemma 5.1 *Let $\mathcal{S}$ be a set of reductions satisfying [CE] and [CS], in an SDRS (in particular, $\mathcal{S}$ may be regular), and let $P : t \xrightarrow{u} s \twoheadrightarrow \; \in \mathcal{S}$. Further:*

(1) *Let v' be a u-residual of $v \in t$, and let v be $\mathcal{S}$-unneeded. Then so is v'.*

(2) *Let u create $v \in s$, and let u be $\mathcal{S}$-unneeded. Then so is v.*

Proof.

(1) *Since v is $\mathcal{S}$-unneeded, there is $Q : t \twoheadrightarrow \; \in \mathcal{S}$ that is external to v. Then Q/u is external to v' by Lemma 2.4, and $Q/u \in \mathcal{S}$ by [CE], Lemma 4.3 and [CS] (indeed, $Q \in \mathcal{S} \wedge Q \trianglelefteq_L u + Q/u$ implies $u + Q/u \in \mathcal{S}$ implies $Q/u \in \mathcal{S}$). Hence v' is $\mathcal{S}$-unneeded.*

(2) *Since u is $\mathcal{S}$-unneeded, there is $Q : t \twoheadrightarrow \; \in \mathcal{S}$ that is external to u. By Lemma 2.6, Q/u is external to v, and $Q/u \in \mathcal{S}$ by [CE] and [CS] (see the proof of (1)). Thus v is $\mathcal{S}$-unneeded.*

□

Definition 5.2 Let $\mathcal{S}$ be a set of reductions in an SDRS $\mathcal{R}$.

(1) We call P *$\mathcal{S}$-quasi-needed* if it contracts infinitely many $\mathcal{S}$-needed redexes.

(2) The *$\mathcal{S}$-needed part of* $P : t \twoheadrightarrow$, written $[P]_{\mathcal{S}}$, is a finite or infinite reduction defined by: $[P]_{\mathcal{S}} = u + [P/u]_{\mathcal{S}}$, where $u \in t$ is the redex whose residual along P is contracted first among $\mathcal{S}$-needed steps in P, if any.

(3) We call $P : t_0 \to t_1 \to \ldots$ *$\mathcal{S}$-needed fair* if for any $\mathcal{S}$-needed redex $v \in t_i$, $v_i \trianglelefteq_{\mathcal{S}} \lceil P \rceil_i$.

It follows immediately from Lemma 5.1 that $[P]_{\mathcal{S}}$ is $\mathcal{S}$-needed. Note that $[P]_{\mathcal{S}}$ is not defined uniquely as its $(k+1)$th step depends on the particular sequentializations of multi-steps (i.e., complete developments of redex sets) in $P/\lceil [P]_{\mathcal{S}} \rceil^k$, but this does not cause any difficulties.

Lemma 5.3 *Let P be a finite or infinite reduction in an SDRS $\mathcal{R}$, and let $\mathcal{S}$ be a regular stable set of reductions in $\mathcal{R}$. Then $[P]_{\mathcal{S}}$ is an $\mathcal{S}$-needed reduction whose length coincides with the number of $\mathcal{S}$-needed steps in P, and $P \approx_{\mathcal{S}} [P]_{\mathcal{S}}$.*

Proof. *Immediate from the fact that, in the notation of Definition 5.2, u has at most one residual along P (by the regularity of $\mathcal{S}$), and one of the residuals of u is contracted in P (see the diagram, where all horizontal steps are $\mathcal{S}$-unneeded and all vertical ones are $\mathcal{S}$-needed).*

$$\begin{tikzcd}
\cdot \arrow[r, two heads] \arrow[d] & \cdot \arrow[d] & & \\
\cdot \arrow[r, two heads] \arrow[d] & \cdot \arrow[r, two heads] & \cdot \arrow[d] & \\
\cdot \arrow[rr, two heads] \arrow[d, "{[P]_{\mathcal{S}}}"'] & & \cdot \arrow[dr, "P"] & \\
{} & & & {}
\end{tikzcd}$$

$\square$

Next we show that, unless it is contracted, an $\mathcal{S}$-needed redexes has at least one $\mathcal{S}$-needed residual. Therefore, residuals of $\mathcal{S}$-quasi-needed reductions remain so. It follows that an $\mathcal{S}$-normalizable term t cannot possess an $\mathcal{S}$-quasi-needed reduction if at least one $\mathcal{S}$-normalizing reduction starting from t is finite.

Lemma 5.4 *Let $\mathcal{S}$ be a regular stable set of reductions in an SDRS $\mathcal{R}$, and let $P : t \xrightarrow{u} s \stackrel{P'}{\twoheadrightarrow} \in \mathcal{S}$. Further, let $u \neq v \in t$, and let v be $\mathcal{S}$-needed. Then v has at least one $\mathcal{S}$-needed residual in s.*

Proof. *Since $P \in \mathcal{S}$ and v is $\mathcal{S}$-needed, $v/u = V \neq \emptyset$. Suppose on the contrary that none of the redexes in V is $\mathcal{S}$-needed. By [CS] and Lemma 5.3, $[P']_{\mathcal{S}} \in \mathcal{S}$, and since $[P']_{\mathcal{S}}$ is $\mathcal{S}$-needed, it is external to V by Lemma 5.1.(1). Hence $u + [P']_{\mathcal{S}}$ is external to v, and $u + [P']_{\mathcal{S}} \in \mathcal{S}$ by [CE], contradicting $\mathcal{S}$-neededness of v.* $\square$

Lemma 5.5 *Let $\mathcal{S}$ be a regular stable set of reductions in an SDRS, let t_0 have an $\mathcal{S}$-quasi-needed reduction, and let $t_0 \xrightarrow{u} s_0$. Then s_0 also has an $\mathcal{S}$-quasi-needed reduction.*

Proof. *By Lemma 5.3, t_0 has an infinite $\mathcal{S}$-needed reduction $P : t_0 \xrightarrow{u_0} t_1 \xrightarrow{u_1} \ldots$ Let $U_i = u/(u_0 + \ldots + u_{i-1})$, $i = 0, 1, \ldots$ (see the diagram below). It follows from finiteness of developments that there are infinitely many numbers k such that $u_k \notin U_k$ (otherwise there would be a number m where $t_m \xrightarrow{u_m} t_{m+1} \xrightarrow{u_{m+1}} \ldots$ is an infinite U_m-development). By Lemma 5.4, u_k has at least one $\mathcal{S}$-needed*

U_k-residual in s_k, i.e. u_k/U_k contains at least one $\mathcal{S}$-needed step. Hence P/u is $\mathcal{S}$-quasi-needed.

$$\begin{tikzcd}
t_0 \arrow[r, "u_0"] \arrow[d, "u = U_0"'] & t_1 \arrow[r, "u_1"] \arrow[d, "U_1"'] & t_2 \arrow[r, two heads] \arrow[d, "U_2"'] & {} \\
s_0 \arrow[r, two heads, "u_0/U_0"'] & s_1 \arrow[r, two heads, "u_1/U_1"'] & s_2 \arrow[r, two heads] & {}
\end{tikzcd}$$

□

The following theorem justifies the stability concept for sets of reductions:

Theorem 5.6 *Let $\mathcal{S}$ be a regular stable set of reductions in an SDRS $\mathcal{R}$, and let t be $\mathcal{S}$-normalizable.*

(1) Any $\mathcal{S}$-needed fair reduction starting from t is $\mathcal{S}$-normalizing.

(2) If $\mathcal{S}$ contains a finite reduction starting from t, then t does not possess a reduction in which $\mathcal{S}$-needed redexes are contracted infinitely many times.

Proof.

(1) By Lemma 5.3, there is an $\mathcal{S}$-needed reduction $P : t \twoheadrightarrow\ \in \mathcal{S}$. Let Q be $\mathcal{S}$-needed fair. By [CE], it is enough to prove that $P \trianglelefteq_{\mathcal{S}} Q$, which requires proving that $\lceil P \rceil^k \trianglelefteq_{\mathcal{S}} Q$ for every finite initial part $\lceil P \rceil^k$ of P. Suppose $\lceil P \rceil^k \trianglelefteq_{\mathcal{S}} Q$, and let us prove $\lceil P \rceil^{k+1} \trianglelefteq_{\mathcal{S}} Q$. Then $\lceil P \rceil^k / \lceil Q \rceil^{l_k}$ is $\mathcal{S}$-unneeded for some l_k. Let v be the $(k+1)$th step of P. If v has an $\mathcal{S}$-needed $\lceil Q \rceil^{l_k} / \lceil P \rceil^k$-residual v', then by Lemma 5.1 v' must be a residual of some $\mathcal{S}$-needed redex v'' in the final term t_{l_k} of Q_{l_k} (see the diagram), and $v'' \trianglelefteq_{\mathcal{S}} \lceil Q \rceil_k$ since Q is $\mathcal{S}$-needed fair. Thus, by the Cube Lemma and Lemma 5.1, $v' \trianglelefteq_{\mathcal{S}} \lceil Q \rceil_{l_k} / (\lceil P \rceil^k / \lceil Q \rceil^{l_k})$, and we are done.

$$\begin{tikzcd}
t_0 \arrow[r, two heads, "\lceil P \rceil^k"] \arrow[d, two heads, "\lceil Q \rceil^{l_k}"'] & \cdot \arrow[r, "v"] \arrow[d, two heads] & {} \\
t_{l_k} \arrow[r, two heads, "\lceil P \rceil^k / \lceil Q \rceil^{l_k}"] \arrow[d, two heads, "\lceil Q \rceil_{l_k}"'] & \cdot \arrow[r, "v'"] \arrow[d, two heads] & {} \\
{} & {} &
\end{tikzcd}$$

(2) Let $P : t \twoheadrightarrow s$ and $P \in \mathcal{S}$. Suppose on the contrary that there is an $\mathcal{S}$-quasi-needed Q starting from t. Then by Lemma 5.5 Q/P is $\mathcal{S}$-quasi-needed as well – a contradiction, since $P \in \mathcal{S}$ implies that Q/P is $\mathcal{S}$-unneeded.

□

Based on the result in section 3, we will now give a sufficient and necessary condition for superstability of regular stable sets of finite results.

Definition 5.7 Let $\mathcal{S}$ be a set of reductions in a DRS.

(1) We call $u \in t$ *$\mathcal{S}$-erased* if u does not have a residual under any $\mathcal{S}$-normalizing reduction.

(2) We call a reduction $\mathcal{S}$-*erased* if it only contracts $\mathcal{S}$-erased redexes.

Clearly, a reduction P is $\mathcal{S}$-erased iff $P \trianglelefteq_L Q$ for every $Q \in \mathcal{S}$. Note that $\mathcal{S}$-erased redexes need not be $\mathcal{S}$-needed (e.g., when $\mathcal{S}$ is the set of normal forms in an orthogonal TRS that has an erasing rule, say $f(x) \to a$). However, any reducible normalizable term t in an orthogonal (first or higher-order) TRS has an *external* redex [17]. And such redexes are both erased and needed w.r.t. to the regular stable sets considered in the literature.

Theorem 5.8 *Let $\mathcal{S}$ be a regular stable set of terms, in an SDRS. Then $\mathcal{S}$ is superstable (i.s, satisfies [Min]) iff any $\mathcal{S}$-normalizable term $t \notin \mathcal{S}$ contains an $\mathcal{S}$-erased $\mathcal{S}$-needed redex.*

Proof. *Let $\mathcal{S}_{red}$ be the set of all $\mathcal{S}$-normalizing reductions starting from t, and let $\sqcap_L \mathcal{S}_{red} : t = t_0 \overset{U_0}{\twoheadrightarrow} t_1 \overset{U_1}{\twoheadrightarrow} \ldots$.*

($\Leftarrow$) By the assumption and the construction of $\sqcap_L \mathcal{S}_{red}$, every multi-step of $\sqcap_L \mathcal{S}_{red}$ contracts an $\mathcal{S}$-needed redex. Hence $\sqcap_L \mathcal{S}_{red} \in \mathcal{S}_{red}$ by Theorem 5.6 and $\sqcap_L \mathcal{S}_{red}$ is $\mathcal{S}$-minimal by Theorem 3.2, which imply [Min].

($\Rightarrow$) Since, by [Min], $\sqcap_L \mathcal{S}_{red}$ is $\mathcal{S}$-normalizing, it contracts a $\mathcal{S}$-needed redex by [CN]. Let i be the smallest number such that U_i contains an $\mathcal{S}$-needed redex, say v_i. Suppose on the contrary that $i \neq 0$. Then, by Lemma 5.1, t_{i-1} contains a $\mathcal{S}$-needed redex v_{i-1} for which v_i is the unique residual (since by the minimality of i, all redexes in U_{i-1} are $\mathcal{S}$-unneeded). But v_{i-1} is erased along $\sqcap_L \mathcal{S}_{red}$ since v_i is, and by the $\mathcal{S}$-minimality of $\sqcap_L \mathcal{S}_{red}$, v_{i-1} is $\mathcal{S}$-erased. Hence we must have $v_{i-1} \in U_{i-1}$ – a contradiction. $\square$

This theorem gives a more 'local' definition of superstability for regular stable sets of finite results. By combining this with the Relative Hypernormalization theorem, we get the following corollary:

Corollary 5.9 *(**Minimal Relative Normalization**) Let $\mathcal{S}$ be a superstable set of terms in an SDRS, and let t be an $\mathcal{S}$-normalizable term not in $\mathcal{S}$. $\mathcal{S}$-minimal $\mathcal{S}$-normalizing reductions arise from repeatedly contracting $\mathcal{S}$-needed $\mathcal{S}$-erased redexes in t. A finite number of $\mathcal{S}$-unneeded but $\mathcal{S}$-erased redexes may also be contracted without losing $\mathcal{S}$-minimality.*

6 Concluding remarks

We have formalized a concept of 'infinite results' for finite terms, and proved a normalization theorem relative to such results, in an abstract framework of SDRSs. We also studied minimal relative normalization in SDRSs. These results unify and extend several earlier results on relative and discrete normalization. Furthermore, concepts of stable sets of results and corresponding orderings introduced in this paper allow SDRSs to be viewed as semi-distributive domains, or 'domains with duplication' [24].

References

[1] Abramsky, S., and C.-H. L. Ong, *Full abstraction in the lazy lambda calculus*, Information and Computation **105** (1993) 159-267.

[2] Antoy, S., and A. Middeldorp, *A Sequential Reduction Strategy*, Theoretical Computer Science **165(1)** (1996) 75-95.

[3] Ariola, Z.M., M. Felleisen, J. Maraist, M. Odersky, P.A. Wadler, *A call-by-need lambda calculus*, in: Proc. 22nd ACM Symp. on Principles of Programming Languages (1995) 233-246.

[4] Barendregt, H.P. "The Lambda Calculus, Its Syntax and Semantics" North-Holland, Amsterdam, (1984).

[5] Barendregt, H.P., J.R. Kennaway, J.W. Klop, M.R. Sleep, *Needed reduction and spine strategies for the lambda calculus*, Information and Computation **75(3)** (1987) 191-231.

[6] Berarducci, A., *Infinite lambda-calculus and non-sensible models.* Logic and Algebra, Lecture Notes in Pure and Applied Mathematics **180** (1996), Marcel Dekker Inc., p. 339-378.

[7] Berry, G., "Modéles complétement adéquats et stables des λ-calculs typés", Thèse de l'Université de Paris VII, 1979.

[8] Berry, G., and J.-J. Lévy, *Minimal and optimal computations of recursive programs*, J. of the Association for Computing Machinery **26** (1979) 148-175.

[9] Boudol, G., *Computational semantics of term rewriting systems.* in: M. Nivat and J.C. Reynolds, eds., "Algebraic methods in semantics", Cambridge University Press, (1985) 169-236.

[10] Church, A., and J.B. Rosser, *Some properties of conversion*, Transactions of American Mathematical Society **39** (1936) 472-482.

[11] Curry, H.B., and R. Feys, "Combinatory Logic, Vol. 1" North-Holland, Amsterdam, (1958).

[12] Glauert, J.R.W., and Z. Khasidashvili, *Relative normalization in deterministic residual structures*, in: Proc. CAAP'96, Lecture Notes in Computer Science, **1059** (Springer, Berlin, 1996) 180-195.

[13] Glauert, J.R.W., J.R. Kennaway and Z. Khasidashvili, *Stable results and relative normalization*, Journal of Logic and Computation **10(3)**, Special Issue: Type Theory and Term Rewriting, F. Kamareddine and J.W. Klop, eds. Oxford University Press (2000) 323-348.

[14] Gonthier, G., J.-J. Lévy and P.-A. Melliès, *An abstract standardisation theorem*, in: Proc. 7th IEEE Symp. on Logic in Computer Science (1992) 72-81.

[15] Hindley, R.J., *An abstract form of the Church-Rosser theorem I*, Journal of Symbolic Logic **34(4)** (1969) 545-560.

[16] Huet, G., Confluent reductions: *Abstract properties and applications to term rewriting systems*, J. of the Association for Computing Machinery **27(4)** (1980) 797-821.

[17] Huet, G., and J.-J. Lévy, *Computations in Orthogonal Rewriting Systems*, in: J.-L. Lassez and G. Plotkin, eds., "Computational Logic, Essays in Honor of Alan Robinson", (MIT Press, 1991) 394-443.

[18] Kennaway, J.R., *Sequential evaluation strategy for parallel-or and related reduction systems*, Annals of Pure and Applied Logic **43** (1989) 31-56.

[19] Kennaway, J.R., J.W. Klop, M.R. Sleep and F.-J. de Vries, *Event structures and orthogonal term graph rewriting*, in: M.R. Sleep, M.J. Plasmeijer, M.C.J.D. van Eekelen, eds., "Term Graph Rewriting: Theory and Practice" (John Wiley, 1993) 141-156.

[20] Kennaway, J.R., J.W. Klop, M.R. Sleep and F.-J. de Vries, *Transfinite reductions in orthogonal term rewriting systems*, Information and Computation **119(1)** (1995) 18-38.

[21] Kennaway, J.R., J.W. Klop, M.R. Sleep and F.-J. de Vries, *Infinitary Lambda Calculus*, Theoretical Computer Science **175(1)** (1997) 93-125.

[22] Khasidashvili, Z., *β-reductions and β-developments of λ-terms with the least number of steps*, in: Proc. COLOG'88, Lecture Notes in Computer Science, **417** (Springer, Berlin, 1990) 105-111.

[23] Khasidashvili, Z., *Optimal normalization in orthogonal term rewriting systems*, in: Proc. RTA'93, Lecture Notes in Computer Science, **690** (Springer, Berlin, 1993) 243-258.

[24] Khasidashvili, Z., *Stable computational semantics of conflict-free rewrite systems.*

[25] Khasidashvili, Z., and J.R.W. Glauert, *Discrete normalization and standardization in deterministic residual structures*, in: Proc. ALP'96, Lecture Notes in Computer Science, **1139** (Springer, Berlin, 1996) 135-149.

[26] Khasidashvili, Z., and J.R.W. Glauert, *Relating conflict-free stable transition and event models (extended abstract)*, in: Proc. MFCS'97, Lecture Notes in Computer Science, **1295** (Springer, Berlin, 1997), 269-278.

[27] Khasidashvili, Z., and J.R.W. Glauert, *Relating conflict-free stable transition and event models via redex families.* Special issue dedicated to MFCS'97 conference, P. Ružička, ed. Theoretical Computer Science (to appear).

[28] Khasidashvili, Z. and J.R.W. Glauert, *An Abstract Böhm-normalization.* Technical Report.

[29] Klop, J.W., *Term rewriting systems*, in: S. Abramsky, D. Gabbay and T. Maibaum, eds., "Handbook of Logic in Computer Science, Vol. 2", (Oxford University Press, 1992), 1-116.

[30] Lévy J.-J., *An algebraic interpretation of the $\lambda\beta K$-calculus; and an application of a labelled λ-calculus.* Theoretical Computer Science **2(1)** (1976), 97-114.

[31] Lévy, J.-J., "Réductions correctes et optimales dans le lambda-calcul", Thèse de l'Université de Paris VII, 1978.

[32] Lévy, J.-J., *Optimal reductions in the Lambda-calculus*, in: J.R. Hindley and J.P. Seldin, eds., "To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism", (Academic Press, 1980), 159-192.

[33] Longo G., *Set theoretic models of lambda calculus: theories, expansions and isomorphisms*, Annals of Pure and Applied Logic, **24** (1983), 153-188.

[34] Maranget L., "La stratégie paresseuse", Thèse de l'Université de Paris VII, 1992.

[35] Melliès, P.-A., "Description Abstraite des Systèmes de Réécriture", Thèse de l'Université Paris VII, 1996.

[36] Melliès, P.-A., *Axiomatic rewriting theory II: The $\lambda\sigma$-calculus enjoys finite normalization cones*, Journal of Logic and Computation **10(3)**, Special Issue: Type Theory and Term Rewriting, F. Kamareddine and J.W. Klop, eds. Oxford University Press (2000) 461-487.

[37] Middeldorp, A., *Call by need computations to root-stable form*, in: Proc. 24th ACM Symp. on Principles of Programming Languages (1997) 94-105.

[38] Nöcker, E., "Efficient functional programming: Compilation and programming techniques", Ph.D. Thesis, Katholic University of Nijmegen, 1994.

[39] van Oostrom, V., "Confluence for abstract and higher-order rewriting", Ph.D. Thesis, Free University, Amsterdam, 1994.

[40] van Oostrom, V., *Normalisation in weakly orthogonal rewriting*, in: proc. RTA'99, Lecture Notes in Computer Science, **1631** (Springer, Berlin, 1999), 60-74.

[41] van Raamsdonk, F., "Confluence and normalisation for higher-order rewriting", Ph.D. Thesis, Free University, Amsterdam, 1996.

[42] Stark, E.W., *Concurrent transition systems*, Theoretical Computer Science, **64(3)** (1989) 221-269.

[43] Winskel, G., A*n introduction to event structures*, in: Proc. Linear time, branching time and partial order in logics and models of concurrency, Lecture Notes in Computer Science, **354** (Springer, Berlin, 1989), 364-397.

[44] Xi, H., *Evaluation under lambda abstraction*, in: proc. PLILP'97, Lecture Notes in Computer Science, **1292** (Springer, Berlin, 1997), 259-274.

[45] Yoshida, N., *Optimal reduction in weak λ-calculus with shared environments*, in: Proc. ACM Conference on Functional Programming Languages and Computer Architecture (1993), 243-252.

Operational Semantics for Lazy Functional Logic Programs ⋆

Elvira Albert[1] Michael Hanus[2] Frank Huch[2]
Javier Oliver[1] Germán Vidal[1]

[1] *DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain*
{ealbert,fjoliver,gvidal}@dsic.upv.es

[2] *Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany*
{mh,fhu}@informatik.uni-kiel.de

Abstract

In this paper we define an operational semantics for lazy functional logic programs including notions like sharing, concurrency, non-determinism, etc. Such a semantic description is not only important to provide appropriate language definitions to reason about programs and check the correctness of implementations but it is also a basis to develop language-specific tools, like program tracers, profilers, optimizers, etc. First, we consider a "big-step" semantics in natural style to relate expressions and their evaluated results. Since this semantics is not sufficient to cover concurrency, search strategies, or to reason about costs associated to particular computations, we define a "small-step" operational semantics covering the features of modern functional logic languages.

---

1 Introduction

This paper is motivated by the fact that there does not exist a precise definition of an operational semantics covering all aspects of modern lazy functional logic languages, like sharing, concurrency, logical variables, non-determinism, higher-order, constraints, etc. For instance, the report on the multi-paradigm language Curry [12] contains a fairly precise operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy [17] is based on narrowing and sharing (without concurrency) but the formal definition is based on a narrowing calculus [9] which does not include a particular pattern-matching strategy. However, the latter becomes

⋆ This work has been partially supported by CICYT TIC 2001-2705-C03-01, by Acción Integrada Hispano-Alemana HA2001-0059, by Acc. Int. Hispano-Austriaca HU2001-0019, by Acc. Int. Hispano-Italiana HI2000-0161, and by the DFG under grant Ha 2457/1-2.

important if one wants to reason about costs of computations (see [5] for a discussion about narrowing strategies and calculi). [13] contains an operational semantics for a lazy narrowing strategy but it addresses neither concurrency nor aspects of search strategies.

In order to provide theoretical foundations for reasoning about programs, correctness of implementations, optimizations, or costs of computations, we define an operational semantics covering the important aspects of current functional logic languages. First, we consider a *natural* semantic description which defines the intended results by relating expressions to values [2]. This "big-step" semantics is non-deterministic and does not cover all aspects (e.g., concurrency, search strategies). However, it accurately models sharing which is important not only to reason about the space behavior of programs (as in [16]) but also for the correctness of computed results in the presence of non-confluent function definitions [9]. Then, we provide a more implementation-oriented semantics based on the definition of individual computation steps, and we extend it to cover concurrency and search strategies. This semantic description is the formal basis to reason about operational aspects of programs, for instance, in order to develop appropriate debugging tools. In particular, it is a basis to provide a comprehensive definition of Curry (in contrast to [10,12] which contain only partial definitions). One can use it to prove the correctness of implementations by further refinements, as done in [21]. Furthermore, one can count the costs (time/space) associated to particular computations in order to justify optimizations [1,4,22] or to compare different search strategies.

The paper is organized as follows. In the next section we introduce some foundations for understanding the subsequent development. Section 3 recalls a semantic description for lazy functional logic programs in natural style. This is refined in Section 4 to a semantics describing individual execution steps. Section 5 shows a deterministic version of the latter in order to make the search strategy explicit. In Section 6, we add concurrency in order to yield the final semantics for declarative multi-paradigm programs. Finally, we discuss in Section 7 an implementation of our semantics and conclude in Section 8 with a comparison to related work.

2 Foundations

A main motivation for this work is to provide theoretical foundations for developing programming tools (like profilers, debuggers, optimizers) for declarative multi-paradigm languages. In order to be concrete, we consider Curry [10,12] as our source language. Curry is a modern multi-paradigm language amalgamating in a seamless way the most important features from functional, logic, and concurrent programming. Its operational semantics is based on an execution model that combines lazy evaluation with non-determinism and concurrency. This model has been introduced in [10] without formalizing the sharing of common subterms. The accurate definition of the latter aspect is one of the purposes of the subsequent sections.

Basically, a Curry program is a set of function definitions (and data definitions for the sake of typing, which we ignore here). Each function is defined by rules describing different cases for input arguments. For instance, the conjunction on Boolean values (`True`, `False`) can be defined by the rules

```
and True  x  =  x
and False x  =  False
```

(data constructors usually start with uppercase and function application is denoted by juxtaposition). Since Curry supports logic programming features, there are no limitations w.r.t. overlapping rules. In particular, one can also have non-confluent rules to define functions that yield more than one result for a given input (these are called *non-deterministic* or *set-valued functions*). For instance, the following function non-deterministically returns one of its arguments as a result:

```
choose x y  =  x
choose x y  =  y
```

A subtle question is the meaning of such definitions if function calls are passed as parameters. Consider, e.g., the definition "`double x = x+x`" and the expression "`double (choose 1 2)`". Similarly to [9], Curry follows the "call-time choice" semantics where all descendants of a subterm are reduced to the same value in a derivation, i.e., the previous expression reduces non-deterministically to one of the values `2` or `4` (but not to `3`). This choice is consistent with a lazy evaluation strategy where all descendants of a subterm are shared [16]. It is our purpose to describe the combination of laziness, sharing, and non-determinism in a precise and understandable manner.

We consider programs where functions are defined by *rules* of the form "$f\,t_1 \ldots t_n$ `=` e" where f is a function, $t_1, \ldots, t_n$ are data terms (i.e., without occurrences of defined functions), the *left-hand side* $f\,t_1 \ldots t_n$ is linear (i.e., without multiple occurrences of variables), and e is a well-formed *expression*.[1] A rule is applicable if its left-hand side matches the current call. Functions are evaluated lazily so that the operational semantics of Curry is a conservative extension of lazy functional programming. It extends the optimal evaluation strategy of [6] by concurrent programming features. These are supported by a concurrent conjunction operator "`&`" on constraints (i.e., expressions of the built-in type `Success`). In particular, a constraint of the form "c_1 `&` c_2" is evaluated by solving both constraints c_1 and c_2 concurrently.

In order to provide an understandable operational description, we assume that programs are translated into a "flat" form, which is a convenient standard representation for functional logic programs. The flat form makes the pattern matching strategy explicit by the use of case expressions, which is important for the operational description; moreover, source programs can be automatically translated into this flat form [11]. The syntax for programs in

[1] Although Curry allows rules with conditions, we will only consider unconditional rules for the sake of simplicity. This is not a real restriction since conditional rules can be translated into unconditional ones by the introduction of auxiliary functions, see [5].

flat form can be summarized as follows:

$$
\begin{array}{llll}
P ::= D_1 \ldots D_m & e ::= x & & \text{(variable)} \\
D ::= f(x_1, \ldots, x_n) = e & \quad \mid & c(e_1, \ldots, e_n) & \text{(constructor call)} \\
 & \quad \mid & f(e_1, \ldots, e_n) & \text{(function call)} \\
p \ ::= c(x_1, \ldots, x_n) & \quad \mid & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(rigid case)} \\
 & \quad \mid & \mathit{fcase}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(flexible case)} \\
 & \quad \mid & e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
 & \quad \mid & \mathit{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathit{in}\ e & \text{(let binding)}
\end{array}
$$

where P denotes a program, D a function definition, p a pattern and e an arbitrary expression. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression e composed by variables (e.g., x, y, z,...), data constructors (e.g., a, b, c,...), function calls (e.g., f, g,...), case expressions, disjunctions (e.g., to represent set-valued functions), and let bindings where the local variables $x_1, \ldots, x_n$ are only visible in $e_1, \ldots, e_n, e$. A case expression has the form[2] $(f)\mathit{case}\ e\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$, where e is an expression, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the corresponding variables of the subexpression e_i. The difference between *case* and *fcase* shows up when the argument e is a free variable: *case* suspends whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression and proceeds with the appropriate branch. Let bindings are in principle not required for translating Curry programs but they are convenient to express sharing without the use of complex graph structures. Operationally, let bindings introduce new structures in memory that are updated after evaluation, which is essential for lazy computations.

As an example, we show the translation of the functions `and` and `choose` into the flat form:

$$
\begin{array}{lcl}
\mathtt{and(x,y)} & = & \mathit{case}\ \mathtt{x}\ \mathit{of}\ \{\mathtt{True} \to \mathtt{y};\ \ \mathtt{False} \to \mathtt{False}\ \} \\
\mathtt{choose(x,y)} & = & \mathtt{x}\ \mathit{or}\ \mathtt{y}
\end{array}
$$

Laziness (or neededness) of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the function's body. In a case expression, the form of the outermost symbol of the case argument is required; therefore, the case argument should be evaluated to a *head normal form* (i.e., a variable or an expression with a constructor at the top). Consequently, our operational semantics will describe the evaluation of expressions only to head normal form. This is not a restriction since the evaluation to normal form or the solving of equations can be reduced to head normal form computations (see [11]). Similarly, the higher-order features of current functional languages can be reduced to first-order definitions by introducing an auxiliary "apply" function [23]. Therefore, we base the definition of our operational semantics on the flat form described above. This is also consistent with current implementations which use the same intermediate language [7].

[2] We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$ and $(f)\mathit{case}$ for either *fcase* or *case*.

Extra variables are those variables in a rule which do not occur in the left-hand side. Such extra variables are intended to be instantiated by constraints in conditions or right-hand sides. They are usually introduced in Curry programs by a declaration of the form `let x free in...` As Antoy [5] pointed out, the use of extra variables in a functional logic language causes no conceptual problem if these extra variables are renamed whenever a rule is applied. We will model this renaming similar to the renaming of local variables in let bindings. For this purpose, we assume that all extra variables x are explicitly introduced in flat programs by a let binding of the form $let\ x = x\ in\ e$. Throughout this paper, we call such variables which are bound to themselves *logical variables*. For instance, an expression $x + y$ with logical variables x and y is represented as $let\ x = x, y = y\ in\ x + y$. Therefore, such a (circular) let binding indicates logical variables in our semantic treatment. It is interesting to note that circular bindings are also used in implementations of Prolog to represent logic variables [24].

3 A Natural Semantics

In this section, we recall a natural (big-step) semantics for lazy functional logic programs [2] which is in the midway between a (simple) denotational semantics and a (complex) operational semantic description for a concrete abstract machine. This semantics is non-deterministic and accurately models sharing. Let us illustrate the effect of sharing by means of an example.

Example 3.1 Consider the following (flat) program:

$$\begin{array}{lcl} \mathtt{foo(x)} & = & \mathtt{addB(x,x)} \\ \mathtt{bit} & = & 0\ or\ 1 \\ \mathtt{addB(x,y)} & = & case\ \mathtt{x}\ of\ \{0 \to \mathtt{y};\ 1 \to case\ \mathtt{y}\ of\ \{0 \to 1;\ 1 \to \mathtt{BO}\}\} \end{array}$$

Under a sharing-based implementation, the computation of "foo(e)" must evaluate the expression e only once. Therefore, the evaluation of the expression `foo(bit)` must return either `0` or `BO` (binary overflow). Note that, without sharing, the results would be `0`, `1`, or `BO`.

The definition of the big-step semantics mainly follows the natural semantics defined in [16] for the lazy evaluation of functional programs. In this (higher-order) functional semantics, the *let* construct is used for the creation and sharing of *closures* (i.e., functional objects created as the value of lambda expressions). The key idea in Launchbury's natural semantics is to describe the semantics in two parts: a "normalization" process—which consists in converting the λ-calculus into a form where the creation and sharing of closures is made explicit—followed by the definition of a simple semantics at the level of closures. Similarly, the (first-order) semantics for lazy functional logic programs is described in two separated phases. In the first phase, a normalization process is applied in order to ensure that the arguments of functions and constructors are always variables (not necessarily different) and that all bound variables are completely fresh variables.

Definition 3.2 The normalization of an expression e proceeds in two stages. First, we flatten all the arguments of function (or constructor) calls by means of the mapping e^*, which is defined inductively as follows:

$$\begin{aligned}
x^* &= x \\
h(x_1, \ldots, x_n)^* &= h(x_1, \ldots, x_n) \\
h(x_1, \ldots, x_{i-1}, e_i, e_{i+1}, \ldots, e_n)^* &= let\ x_i = e_i^*\ in\ h(x_1, \ldots, x_{i-1}, x_i, e_{i+1}, \ldots, e_n)^* \\
&\ \text{where } e_i \text{ is no variable and } x_i \text{ fresh} \\
(let\ \{\overline{x_k = e_k}\}\ in\ e)^* &= let\ \{\overline{x_k = e_k{}^*}\}\ in\ e^* \\
(e_1\ or\ e_2)^* &= e_1{}^*\ or\ e_2{}^* \\
((f)case\ e\ of\ \{\overline{p_k \to e_k}\})^* &= (f)case\ e^*\ of\ \{\overline{p_k \mapsto e_k{}^*}\}
\end{aligned}$$

Here, h denotes either a constructor or a function symbol.

The second stage consists in applying α-conversion in order to have fresh variable names for all bound variables in e. The extension of this normalization process to programs is straightforward.

Normalization introduces one new let construct for each non-variable argument. Trivially, this could be modified in order to produce one single let with the bindings for all non-variable arguments of a function (or constructor) call, which we assume for the subsequent examples.

For the definition of the natural semantics, we consider that both the program and the expression to be evaluated have been previously normalized.

Example 3.3 The normalization of the program and expression of Example 3.1 returns the program unchanged and the following expression:

$$let\ \texttt{x1} = \texttt{bit}\ \ in\ \ \texttt{foo(x1)}$$

The state transition semantics is shown in Figure 1. The rules obey the following naming conventions:

$$\Gamma, \Delta, \Theta \in Heap \ = \ Var \to Exp \qquad\qquad v \in Value \ ::= \ x \mid c(\overline{e_n})$$

where Var and Exp represent the domain of variables and expressions, respectively. Furthermore, we use x, y to denote variable names, t for constructor-rooted terms, and e for arbitrary expressions. A *heap* is a partial mapping from variables to expressions. The *empty heap* is denoted by [ ]. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap). We use judgements of the form "$\Gamma : e \Downarrow \Delta : v$", which should be interpreted as "the expression e in the context of the heap Γ evaluates to the value v with the (modified) heap Δ". Let us briefly explain the rules in Figure 1.

(VarCons). If a variable is bound to a constructor-rooted term in the heap, this rule simply returns the associated term; the heap remains unchanged.

(VarExp). This rule achieves the effect of sharing. If the variable to be evaluated is bound to some expression in the heap, then the expression is evaluated and the heap is updated with the computed value; finally, this value is returned as the result. In contrast to [16], the binding for the variable is not removed from the heap; this becomes useful to generate fresh variable names easily.

$$\text{(VarCons)} \quad \Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t$$

$$\text{(VarExp)} \quad \frac{\Gamma[x \mapsto e] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} \quad \begin{array}{c}\text{where } e \text{ is not constructor-rooted} \\ \text{and } e \neq x\end{array}$$

$$\text{(Val)} \quad \Gamma : v \Downarrow \Gamma : v \quad \begin{array}{c}\text{where } v \text{ is constructor-rooted} \\ \text{or a variable with } \Gamma[x] = x\end{array}$$

$$\text{(Fun)} \quad \frac{\Gamma : \rho(e) \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v} \quad \text{where } f(\overline{y_n}) = e \in \mathcal{R} \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$

$$\text{(Let)} \quad \frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Delta : v} \quad \begin{array}{c}\text{where } \rho = \{\overline{x_k \mapsto y_k}\} \\ \text{and } \overline{y_k} \text{ are fresh variables}\end{array}$$

$$\text{(Or)} \quad \frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ or\ e_2 \Downarrow \Delta : v} \quad \text{where } i \in \{1, 2\}$$

$$\text{(Select)} \quad \frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \Downarrow \Theta : v}{\Gamma : (f)case\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v} \quad \begin{array}{c}\text{where } p_i = c(\overline{x_n}) \\ \text{and } \rho = \{\overline{x_n \mapsto y_n}\}\end{array}$$

$$\text{(Guess)} \quad \frac{\Gamma : e \Downarrow \Delta : x \qquad \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \Downarrow \Theta : v}{\Gamma : fcase\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v}$$
$$\text{where } p_i = c(\overline{x_n}),\ \rho = \{\overline{x_n \mapsto y_n}\},\ \text{ and } \overline{y_n} \text{ are fresh variables}$$

Fig. 1. Natural Semantics for Lazy Functional Logic Programs

(Val). For the evaluation of a value, this rule trivially returns it without modifying the heap.

(Fun). This rule corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule.

(Let). In order to reduce a let construct, this rule adds the bindings to the heap and, then, proceeds with the evaluation of the main argument of the let. Note that the variables introduced by the let construct are renamed with fresh names to avoid variable name clashes.

(Or). It non-deterministically evaluates an *or* expression by either evaluating the first argument or the second argument.

(Select). This rule corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, the appropriate branch is selected and, then, the evaluation of the expression in this branch proceeds by applying the corresponding matching substitution.

(Guess). This rule corresponds to the evaluation of a flexible case expression whose argument reduces to a logical variable. It non-deterministically binds this variable to one of the patterns and proceeds with the evaluation of the corresponding branch. Renaming of pattern variables is also necessary in order to avoid variable name clashes. Additionally, the heap is updated with the (renamed) logical variables of the pattern.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{1} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{1}}\ \mathsf{Val}}{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{0}\ \mathit{or}\ \mathtt{1} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{1}}\ \mathsf{Or}}{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{bit} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{1}}\ \mathsf{Fun}}{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{x1} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{1}}\ \mathsf{VarExp} \qquad \boxed{\mathtt{sub\text{-}proof}}}{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathit{case}\ \mathtt{x1}\ \mathit{of}\ \{0 \to 0;\ 1 \to \mathit{case}\ \mathtt{x1} \ldots\} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0}}\ \mathsf{Select}}{\dfrac{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{addB(x1,x1)} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0}}{\dfrac{[\mathtt{x1} \mapsto \mathtt{bit}] : \mathtt{foo(x1)} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0}}{[\,] : \mathit{let}\ \mathtt{x1} = \mathtt{bit}\ \mathit{in}\ \mathtt{foo(x1)} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0}}\ \mathsf{Let}}\ \mathsf{Fun}}\ \mathsf{Fun}$$

where `sub-proof` has the following form:

$$\dfrac{\dfrac{}{[\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{x1} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{1}}\ \mathsf{VarCons} \qquad \dfrac{}{[\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0}}\ \mathsf{Val}}{[\mathtt{x1} \mapsto \mathtt{1}] : \mathit{case}\ \mathtt{x1}\ \mathit{of}\ \{0 \to 1;\ 1 \to \mathtt{B0}\} \ \Downarrow \ [\mathtt{x1} \mapsto \mathtt{1}] : \mathtt{B0}}\ \mathsf{Select}$$

Fig. 2. Big-step semantics of Example 3.3

A proof of a judgement corresponds to a derivation sequence using the rules of Figure 1. Given a normalized program $\mathcal{R}$ and a normalized expression e (to be evaluated), the *initial configuration* has the form "$[\,] : e$." A derivation is *successful* if it computes a value. If we try to construct a derivation (starting from the left-bottom), then it may *fail* because of two different situations: there may be no finite proof that a reduction is valid—which corresponds to an infinite loop—or there may be no rule which applies in a (sub-part) of the proof. In the latter case, we have two possibilities: either rule Select is not applicable because there is no matching branch or rule Guess cannot be applied because a logical variable has been obtained as the argument of a *rigid* case expression. The natural semantics of Figure 1 does not distinguish between all the above failures. However, they will become observable in the small-step operational semantics of the next section.

Figure 2 illustrates the sharing behavior of the semantic description with one of the possible (non-deterministic) derivations for the program and expression of Example 3.3.

4 A Small-Step Operational Semantics

From an operational point of view, an evaluation in the natural semantics builds a *proof* for "$[\,] : e_0 \ \Downarrow \ \Gamma : e_1$" in a bottom-up manner, whereas a computation by using a small-step semantics builds a sequence of states [21]. In order to transform a natural (big-step) semantics into a small-step one, we need to represent the *context* of sub-proofs in the big-step semantics. For instance, when applying the VarExp rule, a sub-proof for the premise is built. The context (i.e., the rule) indicates that we must update the heap Δ at x with the computed value v for the expression e. This context must be made explicit in the small-step semantics. In our case, the context is *extensible* (i.e., if P' is a sub-proof of P, then the context of P' is an extension of the context

Rule	*Heap*	*Control*	*Stack*
varcons	$\Gamma[x \mapsto t]$	x	S
	$\Longrightarrow \Gamma[x \mapsto t]$	t	S
varexp	$\Gamma[x \mapsto e]$	x	S
	$\Longrightarrow \Gamma[x \mapsto e]$	e	$x : S$
val	Γ	v	$x : S$
	$\Longrightarrow \Gamma[x \mapsto v]$	v	S
fun	Γ	$f(\overline{x_n})$	S
	$\Longrightarrow \Gamma$	$\rho(e)$	S
let	Γ	$let\ \{\overline{x_k = e_k}\}\ in\ e$	S
	$\Longrightarrow \Gamma[\overline{y_k \mapsto \rho(e_k)}]$	$\rho(e)$	S
or	Γ	$e_1\ or\ e_2$	S
	$\Longrightarrow \Gamma$	e_i	S
case	Γ	$(f)case\ e\ of\ \{\overline{p_k \to e_k}\}$	S
	$\Longrightarrow \Gamma$	e	$(f)\{\overline{p_k \to e_k}\} : S$
select	Γ	$c(\overline{y_n})$	$(f)\{\overline{p_k \to e_k}\} : S$
	$\Longrightarrow \Gamma$	$\rho(e_i)$	S
guess	$\Gamma[x \mapsto x]$	x	$f\{\overline{p_k \to e_k}\} : S$
	$\Longrightarrow \Gamma[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}]$	$\rho(e_i)$	S

where in varexp: e is not constructor-rooted and $e \neq x$
val: v is constructor-rooted or a variable with $\Gamma[y] = y$
fun: $f(\overline{y_n}) = e \in \mathcal{R}$ and $\rho = \{\overline{y_n \mapsto x_n}\}$
let: $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ fresh
or: $i \in \{1, 2\}$
select: $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$
guess: $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ fresh

Fig. 3. Non-Deterministic Small-Step Semantics for Functional Logic Programs

of P) and, thus, the representation of the context is made by a *stack*.

A configuration $\Gamma : e$ consists of a heap Γ and an expression e to be evaluated. Now, a *state* (or *goal*) of the small-step semantics is a triple (Γ, e, S), where Γ is the current heap, e is the expression to be evaluated (often called the *control* of the small-step semantics), and S is the stack which represents the current context. *Goal* denotes the domain *Heap* $\times$ *Control* $\times$ *Stack*.

The complete small-step semantics is presented in Figure 3. Let us briefly describe the transition rules. Rule varcons is perfectly analogous to rule VarCons in the natural semantics. In rule varexp, the evaluation of a variable x which is bound to an expression e (which is neither constructor-rooted nor a logical variable) proceeds by evaluating e and, then, adding to the stack the reference to variable x. Here, the stack S is a list and the empty stack is denoted by $[\,]$. When a variable x is on top of the stack, rule val updates the heap with $x \mapsto v$ once a value v is computed. Rules fun, let and or are quite similar to their counterparts in the natural semantics. Rule case initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $(f)\{\overline{p_k \to e_k}\}$ on top of the stack. If we reach a constructor-rooted term, then rule select is used to select the appropriate branch and continue with the evaluation of this branch. If we reach a logical variable, then rule guess is used to non-deterministically choose one alternative and continue with the evaluation of this branch; moreover, the heap is updated with the binding of the logical variable to the corresponding pattern.

In order to evaluate an expression e, we construct an *initial goal* of the form $([], e, [])$ and apply the rules of Figure 3. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $([], e, []) \Longrightarrow^* (\Gamma, e', S)$ is *successful* if e' is in head normal form (the computed *value*) and S is the empty list. The computed *answer* can be extracted from Γ by composing the bindings for the logical variables in the initial expression e. The equivalence of the small-step semantics and the natural semantics is stated as follows [2]:

Theorem 4.1 $([\,], e, [\,]) \;\Longrightarrow^*\; (\Delta, v, [\,])$ *if and only if* $[\,] : e \;\Downarrow\; \Delta : v$.

5 A Deterministic Operational Semantics

The semantics presented in the previous section is still non-deterministic. In actual functional logic languages, this non-determinism is implemented by some search strategy. For debugging or profiling functional logic programs, it is necessary to model search strategies as well. Therefore, we extend the relation $\Longrightarrow$ as follows: $\Longrightarrow \subseteq\; \mathit{Goal} \times \mathit{Goal}^*$. The idea is that a computation step yields a sequence consisting of all possible successor states instead of non-deterministically selecting one of these states. Non-determinism occurs only in the rules or and guess of Figure 3. Thus, the deterministic semantics consists of all rules of Figure 3 except for the rules or and guess which are replaced by the deterministic versions of Figure 4. The main difference is that, in the deterministic versions, all possible successors are listed in the result of $\Longrightarrow$.

With the use of sequences, a search strategy (denoted by "$\circ$") can be defined as a function which composes two sequences of goals. The first sequence represents the new goals resulting from the last evaluation step. The second sequence represents the old goals which must be still explored. For example, a (left-to-right) depth-first search strategy ($\circ_d$) and a breadth-first search strategy ($\circ_b$) can be specified as follows:

$$w \circ_d v = wv \quad \text{and} \quad w \circ_b v = vw$$

A small-step operational semantics (including search) which computes the first leaf in the search tree w.r.t. a search function $\circ$ can be defined as the smallest

Rule	*Heap*	*Control*	*Stack*		$(Heap \times Control \times Stack)^*$
or	Γ	e_1 *or* e_2	S	$\Longrightarrow$	$(\Gamma, e_1, S)\ (\Gamma, e_2, S)$
guess	$\Gamma[x \mapsto x]$	x	$f\{\overline{p_k \to e_k}\} : S$	$\Longrightarrow$	$(\Gamma[x \mapsto \rho_1(p_1), \overline{y_{n_1} \mapsto y_{n_1}}], \rho_1(e_1), S)$
					$\vdots$
					$(\Gamma[x \mapsto \rho_k(p_k), \overline{y_{n_k} \mapsto y_{n_k}}], \rho_k(e_k), S)$

where in guess: $p_i = c_i(\overline{x_{ni}}),\ \rho_i = \{\overline{x_{ni} \mapsto y_{ni}}\}$, and $\overline{y_{ni}}$ fresh

Fig. 4. Deterministic Small-Step Semantics for Functional Logic Programs

relation $\longrightarrow\ \subseteq\ Goal^* \times Goal^*$ satisfying

$$\text{(Expand)}\quad \frac{g \Longrightarrow G}{g\ G' \longrightarrow G \circ G'} \qquad \text{where } g \in Goal \text{ and } G, G' \in Goal^*$$

The evaluation starts with the initial goal $g_0 = ([\,], e_0, [\,])$ where e_0 is the expression to be evaluated. The relation $\longrightarrow$ is deterministic and it may reach four kinds of *final* states:

Solution: when $\Longrightarrow$ does not yield a successor because the first goal is a solution, i.e., it has the form $(\Gamma, v, [\,])$, where v is the computed value. Furthermore, the computed answer can be obtained from the bindings in Γ for the variables of the initial expression e_0.

Suspension: when $\Longrightarrow$ does not yield a successor because the expression of the first goal is a rigid case expression with a logical variable in the argument position. This situation represents a suspended goal and will be discussed in more detail in the next section.

Fail: when $\Longrightarrow$ does not yield a successor because the value in the case expression of the first goal does not match any of the patterns.

No more goals: All goals have been explored and there are no solutions.

In order to distinguish the different situations, we add a label to the relation $\longrightarrow$ which classifies the leaves of the search tree. The label is computed by means of the following function:

$$type(\Gamma, e, S) = \begin{cases} SUCC & \text{if } e = v,\ S = [\,] \\ SUSP & \text{if } e = x,\ S = \{\overline{p_k \to e_k}\} : S',\ \text{and } \Gamma[x] = x \\ FAIL & \text{if } e = c(\overline{y_n}),\ S = (f)\{\overline{p_k \to e_k}\} : S', \\ & \quad \text{and } \forall i = 1, \ldots, k.\ p_i \neq c(\ldots) \\ EXPAND & \text{otherwise} \end{cases}$$

With this function we can now define the complete evaluation of an expression:

$$\text{(Expand)}\ \frac{g \Longrightarrow G}{g\ G' \stackrel{EXPAND}{\longrightarrow} G \circ G'} \qquad \text{(Discard)}\ \frac{g \not\Longrightarrow}{g\ G' \stackrel{type(g)}{\longrightarrow} G'} \qquad \begin{array}{l}(g \in Goal \text{ and} \\ \ G, G' \in Goal^*)\end{array}$$

$$\text{(Expand)}\ \frac{(\Gamma, e, S) \Longrightarrow (\Gamma_1, e_1, S_1) \ldots (\Gamma_n, e_n, S_n)}{(\Gamma, T(e,S)T') : G \overset{EXPAND}{\longrightarrow} (\Gamma_1, T(e_1,S_1)T') \ldots (\Gamma_n, T(e_n,S_n)T') \circ G}$$

$$\text{(Fork)}\ \frac{-}{(\Gamma, T(e_1\&e_2,S)T') : G \overset{EXPAND}{\longrightarrow} (\Gamma, T(e_1,S)(e_2,S)T') \circ G}$$

$$\text{(Fail)}\ \frac{\mathit{type}(\Gamma, e, S) = \mathit{FAIL}}{(\Gamma, T(e,S)T') : G \overset{FAIL}{\longrightarrow} G} \qquad \text{(Succ)}\frac{\forall 1 \leq i \leq k :\ \mathit{type}(\Gamma, e_i, S_i) = \mathit{SUCC}}{(\Gamma, (e_1,S_1)\ldots(e_k,S_k)) : G \overset{SUCC}{\longrightarrow} G}$$

$$\text{(Deadlock)}\ \frac{\begin{array}{c}\forall 1 \leq i \leq k :\ \mathit{type}(\Gamma, e_i, S_i) \in \{\mathit{SUCC}, \mathit{SUSP}\} \\ \text{and}\quad \exists 1 \leq j \leq k :\ \mathit{type}(\Gamma, e_j, S_j) = \mathit{SUSP}\end{array}}{(\Gamma, (e_1,S_1)\ldots(e_k,S_k)) : G \overset{SUSP}{\longrightarrow} G}$$

Fig. 5. Concurrent Semantics for Multi-Paradigm Programs

The (decidable) condition $g \not\Longrightarrow$ of rule Discard means that none of the rules for $\Longrightarrow$ matches. In this case, $\longrightarrow$ does not perform an *EXPAND* step as the following lemma states (it can be shown by a simple case analysis over $\Longrightarrow$):[3]

Lemma 5.1 *If $g_0 \longrightarrow^* g\ G'$ and $g \not\Longrightarrow$, then $type(g) \neq EXPAND$.*

Now, one can extract the information of interest from the set of (possibly infinite) derivations. For example, the set of all solutions is defined by

$$\mathit{solutions}(g_0) = \{g \mid g_0 \longrightarrow^* g\ G \overset{SUCC}{\longrightarrow} G\}.$$

6 Adding Concurrency

Modern declarative multi-paradigm languages like Curry support concurrency. This makes multi-threading with communication on shared logical variables possible. The simplest semantics for concurrency is *interleaving*, which is usually defined at the level of a small-step semantics. The definition of a concurrent natural semantics would be much more complicated because of the additional don't-care non-determinism of interleaving.

For the formalization of concurrency, we extend the expressions and stacks in the goals to sequences of expressions and stacks, i.e., $\mathit{Goal} = \mathit{Heap} \times (\mathit{Control} \times \mathit{Stack})^*$. Each element of $(\mathit{Control} \times \mathit{Stack})^*$ represents a thread and these threads can non-deterministically perform actions (which is the idea of the interleaving semantics). New threads are created with the conjunction operator "&" by extending the sequence with a new thread. The heap is a global entity for all threads in a goal, thus threads communicate with each other by means of variable bindings in this global heap.

The rules for the concurrent semantics are presented in Figure 5, where $T, T' \in (\mathit{Control} \times \mathit{Stack})^*$. The following possibilities for discarding a goal

[3] We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$ including all labels.

are distinguished in the context of the interleaving semantics:

(Fail)	A goal fails if one of its threads fails.
(Succ)	A goal is a solution if all threads terminate successfully.
(Deadlock)	At least one thread suspends and all other threads suspend or succeed.

Our concurrent semantics is *indeterministic* (i.e., don't-care non-deterministic). An evaluation represents one trace of the system. During the evaluation of a goal, several threads may suspend and later be awoken by variable bindings produced by other threads. Then a step with $\Longrightarrow$ is again possible for the awoken process. A goal is only discarded in any of the three cases discussed above. Note that only the *FAIL* case can be interleaved with *EXPAND* steps while in the other two cases there is no alternative successor.

The rule Expand allows computation steps in an arbitrary thread of the first goal. If such a step is don't-know non-deterministic, i.e., yields more than one goal, the entire process structure is copied. Although this is necessary to compute all solutions, it could be more efficient to perform a non-deterministic step only if a deterministic step in another thread is not possible. This strategy corresponds to *stability* in AKL [14] and Oz [20] and could also be specified in our framework without any problem.

We conjecture that $\longrightarrow$ is confluent up to variable renaming because the heap can only be extended. If the variable bindings of different threads in the shared heap clash, then this will happen in any scheduling policy since there is no committed choice construct.

7 Implementation

Our semantic description does not only provide the theoretical foundation to reason about lazy functional logic programs, but it can also be used as a basis to implement interpreters, debuggers, and optimization tools in a high-level manner. In order to get confidence in the latter aspect, we have implemented an interpreter for Curry based on the operational description shown in this paper. The interpreter is written in Haskell [19] and, thus, it can be easily adapted to Curry in order to obtain a meta-interpreter for Curry. The entire implementation consists of a front-end to compile Curry programs into the flat form introduced in Section 2 and an evaluator for expressions based on our small-step semantics. The implementation of the heap uses balanced search trees to ensure efficient access and update operations. In addition to our small-step semantics, the implementation also provides equational constraints and a garbage collector on the heap to execute larger examples. The results are quite encouraging. Standard functional programs are executed (using the Glasgow Haskell compiler) with approximately 24000 reductions per second on a 1.3 GHz Linux-PC (AMD Athlon with 256 KB cache). For logic programs involving search more than 2000 non-deterministic steps are executed per second. Although our interpreter is much slower than compilers based on back-ends implemented in low-level (non-declarative) languages, it is comparable to other meta-interpreters. In particular, it is faster than previous meta-interpreters for Curry (e.g., [3]) due to an improved handling of variable sharing. Thus, our implementation can be an appropriate basis for developing further tools like program optimizers based on partial evaluators, visualization tools, etc.

8 Conclusions and Related Work

We have presented an operational semantics for functional logic languages based on lazy evaluation with sharing, concurrency, and non-determinism implemented by some search strategy. Such a semantic description is important for a precise definition of various constructs of a language, like the combination of set-valued functions, laziness, and concurrency, as well as tools related to operational aspects of a language, like profilers and debuggers. We have developed our semantics in several steps. First, programs are transformed into a normalized form in order to make explicit the pattern matching strategy, common subexpressions, etc. Then, we considered a natural semantics for these normalized programs covering laziness, sharing, and non-determinism. In order to cover concurrency and search strategies, we introduced a non-deterministic small-step semantics and proved its equivalence with the natural semantics. Finally, we refined the small-step semantics to describe the concepts not covered by the first semantics, namely search strategies and concurrency. Thus, the final semantics provides an appropriate foundation to define lazy functional logic programming languages like Curry. Mainly, only the handling of predefined functions needs to be added.

Let us compare our work with some other related approaches. Launchbury [16], Sestoft [21], and Sansom and Peyton Jones [19] gave similar descriptions for purely lazy functional languages where logic programming features and concurrency are not covered. [15] and [8] contain operational and denotational descriptions of Prolog with the main emphasis on covering the backtracking strategy and the “cut” operator. Although the modeling of backtracking by the use of a goal sequence has some similarities with our description, laziness, sharing, and concurrency are not covered there. Podelski and Smolka [18] defined an operational semantics for constraint logic programs with coroutining in order to specify the interaction of backtracking, cut, and coroutining. Their modeling of coroutining via “pools” is related to our model of concurrency, but demand-driven evaluation and sharing is not contained in their semantics. Operational semantics for functional logic programs with sharing can be found in [9] and [13]. However, [9] does not model the pattern-matching strategy used in real implementations, and [13] does not model search strategies and concurrency (but allows partial applications in case patterns).

In order to obtain a complete operational description of a practical language like Curry, one has to add descriptions for solving equational constraints and evaluating external functions and higher-order applications to the semantics presented in this paper. This is subject of ongoing work [2].

For future work, we plan to enhance this operational semantics with the computation of cost information (which is useful, e.g., for profiling [4,19]). Furthermore, it could be interesting to use our operational semantics as a basis to develop debugging and optimization tools (like partial evaluators [3]), and to check or derive new implementations (like in [21]) for Curry.

References

[1] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.

[2] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. An Operational Semantics for Declarative Multi-Paradigm Languages. Technical report, DSIC, Technical University of Valencia (UPV), 2002. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[3] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[4] E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*. Springer LNCS, 2002. To appear.

[5] S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.

[6] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[7] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[8] S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming (5)*, pages 61–91, 1988.

[9] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming based on a Rewriting Logic. *Journal of Logic Programming*, 40:47–87, 1999.

[10] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 80–93. ACM, New York, 1997.

[11] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[12] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.

[13] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 216–232. Springer LNCS 2024, 2001.

[14] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[15] N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming*, pages 281–288, 1984.

[16] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of ACM Principles of Programming Languages*, pages 144–154. ACM Press, 1993.

[17] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[18] A. Podelski and G. Smolka. Operational Semantics of Constraint Logic Programs with Coroutining. In *Proc. of the Twelfth International Conference on Logic Programming (ICLP'95)*, pages 449–463. MIT Press, 1995.

[19] P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, 1997.

[20] C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[21] P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[22] G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press, 2002.

[23] D. H. D. Warren. Higher-Order Extensions to Prolog – Are they needed? In Michie Hayes-Roth and Pao, editors, *Machine Intelligence*, volume 10. Ellis Horwood, 1982.

[24] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical note 309, SRI International, Stanford, 1983.

Pattern-driven Reduction in Haskell

William L. Harrison [1]

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University
Beaverton, Oregon, USA

Richard B. Kieburtz [2]

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University
Beaverton, Oregon, USA

Abstract

Haskell is a functional programming language with nominally non-strict semantics, implying that evaluation of a Haskell expression proceeds by demand-driven reduction. However, Haskell also provides pattern matching on arguments of functions, in **let** expressions and in the match clauses of **case** expressions. Pattern-matching requires data-driven reduction to the extent necessary to evaluate a pattern match or to bind variables introduced in a pattern. In this paper we provide both an abstract semantics and a logical characterization of pattern-matching in Haskell and the reduction order that it entails.

1 Introduction

Haskell is known as a "lazy" functional language, implying that the default reduction strategy used to evaluate terms to a head-normal form is leftmost-outermost. Furthermore, the data constructors of Haskell's algebraic data types are non-strict—a constructor application is a head-normal form even if the argument terms are unnormalized.

However, Haskell also employs pattern-matching to select among alternative branches of a case expression or alternative equations of a function definition. Pattern-matching can force partial evaluation of an argument or a case discriminator, whether or not the value of any variable bound in the

[1] Email: `wlh@cse.ogi.edu`
[2] Email: `dick@cse.ogi.edu`

```
type Name = String
data LS = Lazy | Strict deriving Eq
data P = Pvar Name | Pcondata Name [P] | Ptilde P | Pwildcard          {- Patterns -}
data E = Var Name | Undefined | ConApp (Name,[LS]) [E] | Case E [(P,E)]      {- Expressions -}
```

Fig. 1. Abstract Syntax of a Haskell Fragment

pattern is ever demanded. In that sense, the reduction order mandated by pattern-matching is not lazy and may deviate from a leftmost-outermost strategy.

Of course, situations arise in which a programmer "knows" that a certain pattern will occur in an argument, so pattern-matching for control purposes can be shortcut. Haskell provides a prefix notation ($\sim$) with which a programmer can indicate that pattern-matching is disabled for control. A control-disabled pattern (called *irrefutable* in the Haskell literature) serves only to specify that variables occurring in the pattern are bound by a successful match. These bindings can be evaluated lazily. Control-disabled patterns call for a reduction strategy different from nominal, pattern-driven reduction.

The reduction strategy employed to evaluate Haskell programs containing pattern constructs is implicit in the language definition, although a careful reading is necessary. It is perhaps the language aspect least well understood by Haskell amateurs. This paper gives a semantics for the fragment of Haskell that involves pattern-matching. It also proposes inference rules of a programming logic for this language fragment. The semantics and the logic provide complementary but coherent formal descriptions that are consistent with the language definition [4]. They should help a reader to comprehend the nuances of Haskell reduction strategies.

2 A Haskell fragment and its informal semantics

2.1 Patterned abstractions

Patterns may occur in several different syntactic contexts in Haskell—in case branches, explicit abstractions, or on the left-hand sides of definitions. We say that a pattern is *abstracted* if it occurs in an operand position on the left-hand side of a function definition [3], under a lambda-symbol (the backslash, in Haskell) or to the left of the arrow symbol (->) in a case branch. Since the roles played by abstracted patterns are similar in every context, we shall focus on patterns as they occur in case expressions.

2.1.1 Evaluating case expressions

When a case expression is evaluated, the first case branch is applied to the case discriminator (the expression between the keywords case... of). If the

[3] In a local (let) definition, a pattern may occur as the entire left-hand side of an equation. Such an occurrence is implicitly control-disabled, even if it is not prefixed by the character ($\sim$).

case discriminator matches the abstracted pattern of this branch, then the body of the case branch is evaluated in a context extended with the value bindings of pattern variables made by the match. If the discriminator fails to match the pattern, then the next in the list of case branches is applied to the discriminator, and so on, until a pattern is found to matche. If there is no branch whose pattern matches, then evaluation of the case expression fails with an unrecoverable error.

2.1.2 Matching abstracted patterns

An abstracted pattern fulfills two roles:

- **Control**: A `case` discriminator expression is evaluated to the extent necessary to determine whether it matches the pattern of a case branch. If the match fails, control shifts to try a match with the next alternative branch, if one is available.
- **Binding**: When a match succeeds, each variable occurring in the pattern is bound to a subterm corresponding in position in the (partially evaluated) `case` discriminator. Since patterns in Haskell cannot contain repeated occurrences of a variable, the bindings are unique at any successful match.

2.1.3 Variables and wildcard patterns

A variable is itself a pattern which matches any term[4]. Thus a match with a variable never fails and always accomplishes a binding. A term need not be evaluated to match with a pattern variable.

Haskell designates a so-called wildcard pattern by the underscore character (_). The wildcard pattern, like a variable, never fails to match but it entails no binding.

2.1.4 Constructor patterns: strict and lazy

When a data constructor occurs in a pattern, it must appear in a *saturated* application to sub-patterns. That is, a constructor typed as a k-ary function in a datatype declaration must be applied to exactly k sub-patterns when it is used in a pattern.

When a constructor occurs as the top-level operator in a pattern, a match can occur only if the `case` discriminator evaluates to a term that has the same constructor as its primary operator. Subterms of the discriminator must match the corresponding sub-patterns of the constructor pattern or else the entire match fails. If a sub-pattern happens to be a variable or a wildcard, no further evaluation of the corresponding sub-term of the matching expression is required.

However, a constructor may be declared (in a datatype declaration) to be *strict* in one or more of its argument positions by prefixing the character (!) to the type expressions in these argument positions. When a constructor is strict

[4] As Haskell is strongly typed, a variable can only be compared with terms of the same type.

```
-- Semantic Functions for E and P
mE  :: E -> Env -> V
mP  :: P -> V -> Maybe [V]
```

```
-- Environments
type Name = String
type Env = Name -> V
```

```
-- Domain of Values   {- functions -}      {- structured data -}
         data V =      FV (V -> V)     |     Tagged Name [V]
```

```
-- Function composition (diagrammatic)
(>>>) :: (a -> b) -> (b -> c) -> a -> c
f >>> g = g . f
```

```
-- Kleisli composition (diagrammatic)
(<>) :: (a->Maybe b)->(b->Maybe c)-> a->Maybe c
f <> g = \ x -> f x >>= g
```

```
-- Domains are pointed
bottom :: a
bottom = undefined
```

```
-- Purification: the "run" of Maybe monad
purify :: Maybe a -> a
purify (Just x) =  x
purify Nothing  = bottom
```

```
-- Alternation
fatbar :: (a->Maybe b) ->
             (a->Maybe b) -> (a->Maybe b)
f `fatbar` g = \ x -> (f x) `fb` (g x)
    where fb :: Maybe a -> Maybe a -> Maybe a
          Nothing `fb` y = y
          (Just v) `fb` y = (Just v)
```

```
-- Semantic "seq"
semseq :: V -> V -> V
semseq x y = case x of
                  (FV _)       -> y ;
                  (Tagged _ _) -> y
```

Fig. 2. Semantic Operators

in its i^{th} argument position, an application of the constructor will evaluate its i^{th} argument to head normal form. Thus a pattern match involving a constructor declared to be strict in one or more argument positions implicitly forces evaluation of the corresponding subexpressions of the matching term.

2.1.5 Control-disabled patterns

Disabling a pattern for control does not disable the binding function of a match, it merely defers binding until further computation demands a value for one of the variables occurring in the pattern. When that happens, the focus of computation returns to the deferred pattern match, which is fully computed in order to bind the variables introduced in the pattern. Should a deferred pattern match fail, no alternative is tried, as might have been the case in a normal match failure. Failure of a deferred pattern match causes an unrecoverable program error.

3 Formal Semantics

This section outlines the formal semantics of the Haskell fragment considered in this paper. This semantics has been described in detail elsewhere [3], so the presentation here will be brief. The semantics is presented as a metacircular interpreter for the Haskell fragment whose abstract syntax is given in Figure 1. The interpreter, written in Haskell itself, makes use of standard techniques and structures from the denotational description of programming languages and uses monads to model pattern-matching.

Although the semantic metalanguage here *is* Haskell, care has been taken to use notation which will be recognizable by any functional programmer. However unlike many functional languages, Haskell has explicit monads, and

```
mP  :: P -> V -> Maybe [V]
mP (Pvar x) v                        = Just [v]
mP (Pcondata n ps) (Tagged t vs)     = if n==t then (stuple (map mP ps) vs) else Nothing
mP Pwildcard v                       = Just []
mP (Ptilde p) v = Just(case mP p v of { Nothing -> replicate lp bottom ; Just z -> z })
       where lp = length (fringe p)
             replicate 0 x = []
             replicate n x = x : (replicate (n-1) x)

fringe :: P -> [Name]                --- the fringe of a pattern
fringe (Pvar n)       = [n]          fringe Pwildcard      = []
fringe (Ptilde p)     = fringe p     fringe (Pcondata _ ps) = concat (map fringe ps)

stuple :: [V -> Maybe [V]] -> [V] -> Maybe [V]
stuple [] []        = Just []
stuple (q:qs) (v:vs) = do { v' <- q v ; vs' <- stuple qs vs ; Just (v'++vs') }
```

Fig. 3. Semantics of a Haskell Fragment: Patterns

so we give an overview here of Haskell's monad syntax [5]. The semantics makes use of an error monad [6], which is modeled in Haskell by the `Maybe` monad. This monad is based upon the datatype `Maybe a` as given below, in which the constructor `Just` encloses a normal expression of type `a` and the constructor `Nothing` models an error condition as a data value.

In every monad there is a unit operator (called `return` in Haskell) and a binary operation which is called "bind" in Haskell and represented by the infix operator `(>>=)`. The unit of a monad injects an ordinary (non-monadic) value into the structure of the monad. The bind operation is like a function application expressed in diagrammatic order (i.e. *arg* `>>=` *fn*). It accounts for propagation of the monadic structure of the argument through the computation of the application, which may affect that structure. The unit and bind operators satisfy a set of equations that hold for all monads, thus providing a uniform algebraic framework in which to express computational effects.
The structure of Haskell's `Maybe` monad is specified by:

```
data Maybe a = Just a | Nothing
return :: a -> Maybe a              (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return = Just                       (Nothing >>= f) = Nothing
                                    (Just x >>= f)  = f x
```

Haskell provides an alternative syntax for bind `(>>=)` called "`do` notation" which is defined by:

```
do { y <- x ; f } = (x >>= (\y -> f))
```

Figure 2 contains a description of the semantic setting for the Haskell fragment considered in this paper. It is in most respects a conventional denotational-style semantics for a functional language. The dynamic semantics for expressions, `mE`, maps an expression and an environment to a value in the domain of values `V`. This domain `V` is itself structured conventionally as a universal domain construction [2] over the sum of functions and structured data. The semantics of patterns, `mP`, takes a pattern and returns a map of type `(V->Maybe [V])` (more will be said about this type in Section 3.1). We use

[5] We assume the reader has some familiarity with monads [6].

two composition operators, both written as infix operators in diagrammatic order. The symbol (`>>>`) denotes function composition and the symbol (`<>`) denotes Kleisli composition in the *Maybe* monad. It is assumed that the domain is *pointed* in every type; that is, every domain contains a bottom element `bottom` (usually written $\perp$), which is modeled in Haskell by the polymorphic constant `undefined`.

Figure 2 also displays two combinators integral to modeling `case` expressions and patterns, called `fatbar` and `purify`. If `m1` and `m2` have type `(V->Maybe V)`, then `((fatbar m1 m2) v)` exhibits sequencing behavior similar to `(case v of { m1 ; m2 })`. The `purify` operator converts a `Maybe`-computation into a value, sending a `Nothing` to `bottom`. Post-composing with `purify` signifies that expressions whose evaluation produces certain pattern-match failures (e.g., exhaustion of the branches of a `case` expression) ultimately denote `bottom`.

3.1 Dynamic Semantics of Haskell Pattern Fragment

In the evaluation of (`case` e $\{p_1$ `->` $e_1; \ldots; p_n$ `->` $e_n\}$), patterns p_i are matched against e in left-to-right order until a successful match p_s is found. Then, the value of the whole expression is the value of the expression e_s. If no such match is found, then the value of the whole expression is undefined. For example, evaluating the following `case` expressions gives these results:

```
data Tree = T Tree Tree | S Tree | L | R
case T L R of {T (S x) y -> y; T x y -> x}     ---> L
case T L R of {T ~(S x) y -> y; T x y -> x}    ---> R
case T L R of {T ~(S x) y -> x; T x y -> y}    ---> program error (match)
case T L R of {~(T (S x) y) -> y; T x y -> x}  ---> program error (match)
```

In the first of the `case` expressions above, the constructor `L` fails to match the embedded pattern `(S x)` in the first case branch. The match failure shifts control to the second case branch. In the second example, the embedded pattern `~(S x)` is control-disabled. The term `(T L R)` thus matches the pattern `(T ~(S x) y)` binding `R` to the variable `y`. In the third example, the body of the first case branch demands a value for `x`, thereby forcing a deferred match of the subterm `L` with the pattern `~(S x)`. The deferred match fails, resulting in a program error. The fourth example illustrates that a deferred match of the term `(T L R)` against the pattern `(T (S x) y)` fails, although the match was evaluated in response to a request for a binding for `y` alone. Generally, the effect of deferring pattern match failure is characterized by the following equivalence: `(mP (~p) v)` is `Just[bottom,...,bottom]` $\Leftrightarrow$ `(mP p v)` is `Nothing`. Even when `(mP p v)` fails (i.e., is `Nothing`), `(mP (~p) v)` still succeeds, but all of the bindings created thereby are `bottom`.

4 Logic

While the dynamic semantics defines a meaning for expressions by providing an abstract evaluation model, a verification logic expresses static assertions

```
mE  :: E -> Env -> V
mE (Var n) rho              = rho n
mE (Case e ml) rho          = mcase rho ml (mE e rho)
mE (ConApp (n,ls) es) rho   = evalL (zip es ls) rho n []
   where evalL :: [(E,LS)] -> Env -> Name -> [V] -> V
         evalL [] rho n vs              = Tagged n vs
         evalL ((e,Strict):es) rho n vs = semseq (mE e rho)
                                            (evalL es rho n (vs ++ [mE e rho]))
         evalL ((e,Lazy):es) rho n vs   = evalL es rho n (vs ++ [mE e rho])
mE Undefined rho      = bottom

match :: Env -> (P,E) -> V -> Maybe V
match rho (p,e) = mP p <> ((\vs -> mE e (extend rho xs vs)) >>> Just)
     where xs = fringe p

mcase :: Env -> [(P,E)] -> V -> V
mcase rho ml = (fatbarL $ map (match rho) ml) >>> purify
                     where fatbarL :: [V -> Maybe V] -> V -> Maybe V
                           fatbarL ms = foldr fatbar (\ _ -> (Just bottom)) ms
```

Fig. 4. Semantics of a Haskell Fragment: Expressions

about properties of the semantics. An assertion can take the form of a k-ary predicate applied to k terms. For simplicity, we restrict ourselves here to unary predicates ($k = 1$).

We write t ::: P for the assertion that term t satisfies predicate P. Because function and data constructor applications are non-strict in Haskell's evaluation semantics, two notions of satisfaction of a predicate are sensible.

We say that a predicate, P, is *weakly* satisfied by an expression t if the denotation of t belongs to the set specified by P. It is *strongly* satisfied if, in addition, the denotation of t is not the bottom element in its type domain. By convention, a predicate assumes its weak interpretation unless otherwise annotated. An otherwise weak predicate may be explicitly strengthened by prefixing the symbol ($).

P-logic [6], a verification logic for Haskell, is based upon the familiar Gentzen-style sequent calculus [1]. In this section we give a brief introduction to the syntax of *P*-logic, as well as some inference rules that are relevant to pattern-matching in Haskell.

4.1 Predicates in P-logic

Atomic, unary predicates include the predicate constants, Univ and UnDef, which are respectively satisfied by all terms and by only those terms whose values are undefined.

There are two principal ways that compound predicates are formed in *P*-logic.

(i) The constructors of datatypes declared in a Haskell program are implicitly "lifted" to act as predicate constructors in *P*-logic. For example, in the context of a program, the list constructor (:) combines an expres-

[6] The name *P*-logic is short for *Programatica logic*, as the logic has been developed as part of the Programatica project [5].

sion h of type a and an expression t of type `[`a`]` into a new expression $(h : t)$ of type `[`a`]`. In the context of a formula, the same constructor combines a predicate P and a predicate Q into a new predicate, $(P : Q)$. This predicate is satisfied by a Haskell expression that normalizes to a term of the form $(h : t)$ and whose component expressions weakly satisfy the assertions $h ::: P$ and $t ::: Q$. The default mode of interpretation of the component predicates is weak because the semantics of the data constructor does not require evaluation of its arguments.

(ii) The "arrow" predicate constructor is used to compose predicates that express properties of case branches. An arrow predicate $P \rightarrow Q$ is satisfied by a case branch (p `->` e) if, whenever the case discriminator satisfies the pattern predicate, P, the body, e, of the case branch satisfies Q.

4.2 Inference Rules for Properties of the Haskell Fragment

4.2.1 Constructor application

Rules for constructor application are derived from a Haskell datatype declaration. A datatype declaration serves to define the data constructors of the type, giving the signature of each constructor as a sequence of type expressions.

$$\mathsf{data}\ T = \cdots\ \mid C^{(k)}\, \tau_1 \ldots \tau_k\ \mid\ \cdots$$

Notice that the strictness annotations from the signature of a constructor are represented explicitly in the abstract syntax of a constructor application, although they are not manifested in the concrete syntax.

A constructor application is lifted to a predicate constructor application by the function:

```
sigma :: E -> [Pr] -> Pr
sigma (ConApp (n,ls) _) prs =
   let prs' = take (length ls) prs
       s = and $ map (\(pr,l) -> isStrong pr || l==Lazy) (zip prs' ls)
           where isStrong (Strong _) = True
                 isStrong _ = False
   in if s then Strong $ ConPred (n,ls) prs'
      else ConPred (n,ls) prs'
```

where *ls* lists the strictness declaration (`Lazy` or `Strict`) of the constructor in each argument position. Notice that when a predicate constructor lifted from a strict data constructor is applied to a predicate argument, the resulting predicate is strong only if the argument predicate is so, whereas a predicate derived from a lazy data constructor is always strong. A strong predicate formula $\$C^{(k)}\, P_1 \ldots P_k$ is satisfied by a well-defined term of the form $C^{(k)}\, t_1 \ldots t_k$ whenever each of the t_j satisfies the corresponding predicate P_j.

The rule schemas for properties of a constructor application are:

$$\text{(1)} \qquad \frac{\Gamma \vdash_{\mathcal{P}} t_1 ::: P_1 \ \cdots\ \Gamma \vdash_{\mathcal{P}} t_k ::: P_k}{\Gamma \vdash_{\mathcal{P}} C^{(k)}\, t_1 \ldots t_k ::: C^{(k)}\, P_1 \ldots P_k} \quad (1 \leq k)$$

```
-- Abstract syntax of predicates
data Pr = Univ | UnDef | PrCon Name [Pr] | Strong Pr

-- Skeleton of a pattern
skeleton :: P -> (Name -> Pr) -> Bool -> (Pr,Bool)
skeleton (Pvar n) e b = case (e n) of
                          pr@(Strong _) -> (pr,True)
                          pr            -> (pr,False)
skeleton Pwildcard e _ = (Univ,False)
skeleton (Pcondata (n,ls) ps) e b =
      let (prs,s) = mapSkel ps e True
      in if b || s then (Strong (PrCon n prs),True)
         else (Univ,False)
skeleton (Ptilde p) e _ = skeleton p e False

mapSkel :: [P] -> (Name -> Pr)
             -> Bool -> ([Pr],Bool)
mapSkel [] _ _ = ([],False)
mapSkel (p:ps) e b =
   let (pr,s) = skeleton p e b
       (prs,s') = mapSkel ps e b
   in ((pr:prs),s || s')
```

Fig. 5. Calculation of Pattern Predicates

and, where C and K are distinct constructors in the same data type:

(2)
$$\frac{}{\Gamma \vdash_{\mathcal{P}} C^{(k)}\, t_1 \ldots t_k ::: \$\neg K^{(n)}\ \underbrace{\mathsf{Univ} \ldots \mathsf{Univ}}_{n-times}}$$

4.3 Pattern matching

Match clauses have associated with them predicates of a distinct kind. A match clause whose expression body has the Haskell type (τ) may satisfy a predicate of type *Maybe_Pred* τ. These predicates are formed either with the unary predicate constructor *Just* or the nullary constructor *Nothing*.

4.3.1 Pattern predicates

Because patterns may be nested to arbitrary depths, it is inconvenient to use the syntax of patterns directly in rules. Instead we define a syntactically flattened representation for patterns to allow a simpler representation of pattern predicates in rules.

First, we need the concept of the *fringe* of a pattern, the variables that have defining occurrences in the pattern, listed from left to right.

Definition 4.3.1.1: The *fringe* of a pattern is the list of (distinct) variables whose definition is given by induction on the abstract syntax of patterns by the Haskell function `fringe` in Figure 5.

Next, we define a function that produces a pattern predicate from a pattern and an environment that binds predicates to the variables in the fringe of the pattern. Note that

$$\mathsf{fst}(x, y) = x \text{ and } \mathsf{zip}\,[a_1, \ldots, a_n]\,[b_1, \ldots, b_n] = [(a_1, b_1), \ldots, (a_n, b_n)]$$

Definition 4.3.1.2: The *pattern predicate* formed by instantiating a pattern relative to a predicate environment is defined inductively by the Haskell function `skeleton` in Figure 5.

We use the notation $\pi(p)$ to designate a "flattened" pattern predicate constructor. Formally,

$$\pi(p)\, P_1 \cdots P_n =_{def} \mathsf{fst}\ (skeleton\, p\ (\mathsf{zip}\ (fringe\, p)\ [P_1, \ldots, P_n])\ \mathsf{True})$$
$$\text{where } n = length\ (fringe\, p)$$

□

The pattern predicate calculated by *skeleton* will ignore control-disabled subpatterns that occur in a pattern, replacing them by the universal predicate, Univ, *unless* an explicitly strengthened predicate is bound in the environment to a variable in the fringe of such a subpattern. In such a case, the skeleton of the subpattern is fully elaborated in the *skeleton* computation. In consequence, if an instance of a verification rule such as Rule (3) uses strong predicates in its hypotheses, then the pattern predicate in its conclusion will require a pattern match that evaluates all subterms that are asserted by the strong predicates to be well-defined.

For example, two pattern predicates that are derived from one of the patterns given in the example of Section 3.1 are:

$$\pi(\texttt{T}\ \sim\!\texttt{(S x) y})\ \mathsf{Univ}\ \mathsf{Univ} = \$(\texttt{T}\ \mathsf{Univ}\ \mathsf{Univ})$$
$$\pi(\texttt{T}\ \sim\!\texttt{(S x) y})\ \$\mathsf{Univ}\ \mathsf{Univ} = \$(\texttt{T}\ \$(\texttt{S}\ \$\mathsf{Univ})\ \mathsf{Univ})$$

The strength annotation on the first predicate argument in the second line above forces the pattern predicate to assert definedness of the subpattern `(S x)`.

4.3.2 The domain of a pattern

Informally, the *domain* of a pattern is the set of terms that match the pattern. The criterion for matching patterns in Haskell is complicated somewhat by the possibility that a control-disabled subpattern may be embedded into a normally stricter host pattern. In program execution, the match of a control-disabled pattern that is embedded in a case branch is deferred, pending evaluation of the body of the case branch. The match is dynamically performed only if the body is strict in a variable that occurs in the pattern. When a match failure occurs during a deferred pattern match, the match failure is unrecoverable.

We define the domain of a pattern as a predicate characterizing the set of terms matching the pattern in an non-deferred match.

Definition 4.3.2.1: $Dom(p)$, is the predicate defined by applying the predicate pattern constructor derived from a pattern, p, to a list of Univ predicates.

$$Dom(p) =_{def} \pi(p)\ \mathsf{Univ} \cdots \mathsf{Univ}$$

The domain predicate of a pattern is calculated by:

```
dom p = fst (skeleton p (\_ -> Univ) True)
```

□

Notice that $Dom(p)$ is either Univ or it is a strong predicate.

The formula $\neg Dom(p)$ asserts that a term fails to match p or is undefined. A strengthened domain predicate disjoined with its strong complement is in effect, a partial definedness predicate. Any term that satisfies either $Dom(p)$ or $\$\neg Dom(p)$ is well defined in every subterm necessary to evaluate a control-enabled match with the pattern p.

4.3.3 Properties of case branches

A case branch has a pair of properties, one that it exhibits when a case discriminator matches its pattern and another that characterizes its behavior when pattern-matching fails:

$$\frac{\Gamma,\, x_1 ::: P_1, \cdots, x_n ::: P_n \vdash_{\mathcal{P}} t ::: Q}{\Gamma \vdash_{\mathcal{P}} \{p \;\texttt{->}\; t\} \;:::\; \pi(p)\; P_1 \cdots P_n \rightarrow \$Just\ Q} \tag{3}$$

where $[x_1, \ldots, x_n] = \mathit{fringe}\, p$, and

$$\Gamma \vdash_{\mathcal{P}} \{p \;\texttt{->}\; t\} \;:::\; \$\neg Dom(p) \rightarrow \$Nothing \tag{4}$$

4.3.4 Properties of case expressions

The basic rules for a case expression are those for a single case branch:

$$\frac{\Gamma \vdash_{\mathcal{P}} d ::: \pi(p)[P_1, \ldots, P_k] \qquad \Gamma \vdash_{\mathcal{P}} br \;:::\; \pi(p)\; P_1 \cdots P_n \rightarrow \$Just\ Q}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{br\} ::: \$Just\ Q} \tag{5}$$

$$\frac{\Gamma \vdash_{\mathcal{P}} d ::: \$\neg Dom(p)}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{p \;\texttt{->}\; t\} ::: \$Nothing} \tag{6}$$

The following rules account for a case expression in which multiple case branches are listed.

$$\frac{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{br\} ::: \$Nothing \qquad \Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{brs\} ::: \$Q}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{br;\ brs\} \;:::\; \$Q} \tag{7}$$

where Q has the kind *Maybe_Pred*.

$$\frac{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{br\} \;:::\; \$Just\ P}{\Gamma \vdash_{\mathcal{P}} \mathsf{case}\ d\ \mathsf{of}\ \{br;\ brs\} \;:::\; \$Just\ P} \tag{8}$$

Two rules relate a property of a Haskell term of kind *Maybe Pred* to a property of kind *Pred*.

$$\frac{\Gamma \vdash_{\mathcal{P}} t \;:::\; \$(Just\ P)}{\Gamma \vdash_{\mathcal{P}} t \;:::\; P} \qquad\qquad \frac{\Gamma \vdash_{\mathcal{P}} t \;:::\; \$Nothing}{\Gamma \vdash_{\mathcal{P}} t \;:::\; \mathsf{Univ}} \tag{9}$$

These rules allow a property of a case expression to be propagated to a context that expects a property of kind *Pred*. When a pattern match can be proven to fail, the concluded property, $t ::: \mathsf{Univ}$, provides no specific information.

Figure 6 presents a sample derivation demonstrating how P-logic distinguishes pattern-matching success and failure. The second proof involves a case expression which generates a pattern match failure. Interestingly, the strongest property derivable of this expression in P-logic is Univ.

$$
\dfrac{\dfrac{\dfrac{}{\vdash \mathtt{L} ::: \mathsf{Univ}} \qquad \dfrac{}{\vdash \mathtt{R} ::: \$\mathsf{R}}\,{}^{(1)}}{\vdash (\mathtt{T\,L\,R}) ::: \$(\mathsf{T\,Univ}\,\$\mathsf{R})}\,{}^{(1)} \qquad \dfrac{\dfrac{}{\mathtt{x} ::: \mathsf{Univ}, \mathtt{y} ::: \$\mathsf{R} \vdash \mathtt{y} ::: \$\mathsf{R}}}{\vdash \{(\mathtt{T}{\sim}(\mathtt{S\,x})\,\mathtt{y})\ \texttt{->}\ \mathtt{y}\} ::: \$(\mathsf{T\,Univ}\,\$\mathsf{R}) \rightarrow \$\mathsf{Just}\,\$\mathsf{R}}\,{}^{(3)}}{\dfrac{\vdash \mathtt{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{\ (\mathtt{T}{\sim}(\mathtt{S\,x})\,\mathtt{y})\ \texttt{->}\ \mathtt{y}\ \} ::: \$\mathsf{Just}\,\$\mathsf{R}}{\vdash \mathtt{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{\ (\mathtt{T}{\sim}(\mathtt{S\,x})\,\mathtt{y})\ \texttt{->}\ \mathtt{y}\ \} ::: \$\mathsf{R}}\,{}^{(9)}}\,{}^{(5)}
$$

$$
\dfrac{\dfrac{\dfrac{}{\vdash (\mathtt{T\,L\,R}) ::: \$\neg(\mathsf{S\,Univ})}\,{}^{(2)}}{\vdash \mathtt{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{(\mathtt{S\,x})\ \texttt{->}\ \mathtt{x}\} ::: \$\mathsf{Nothing}}\,{}^{(6)}}{\vdash \mathtt{case}\ (\mathtt{T\,L\,R})\ \mathtt{of}\ \{(\mathtt{S\,x})\ \texttt{->}\ \mathtt{x}\} ::: \mathsf{Univ}}\,{}^{(9)}
$$

Fig. 6. Distinguishing Pattern Match Success from Failure in the Logic
(Numbers refer to the rule that applies at each step.)

5 Conclusion

We have presented a two succinct formalisms that specify the reduction semantics of Haskell pattern-matching, a surprisingly complex aspect of the language. The deferred matching that is required of control-disabled ($\sim$) patterns is of particular interest. In the dynamic semantics, expressed as a meta-language program in Haskell, a deferred pattern match is embedded in a continuation that substitutes the value `bottom` in the bindings of pattern variables in case the match fails. In the verification logic, a pattern predicate is calculated from the pattern and the predicates assumed to be necessary to prove a property of the body of a case branch. This also ensures that a proof of properties will discriminate between failure of a deferred match and failure of a normal match.

In retrospect, one may question whether the generalization of control-disabled patterns, allowing ($\sim$) annotations to be embedded in a pattern in any context, has been worth the complexity that it adds to understanding the operational semantics of Haskell. Nevertheless, in formulating a programming logic for an existing language one must deal with what is there, and that we have done.

The dual development of a denotational semantics and a verification logic for a programming language offers the opportunity to check soundness of the logical inference rules relative to the model provided by the semantics. When the semantics is also given in an executable framework, as is the semantics of Haskell, soundness checking can be partially automated, but a discussion of that topic is left to another paper.

Acknowledgment The authors wish to thank their colleagues on the Programatica project, particularly Jim Hook, Mark Jones and Sylvain Conchon for their encouragement and for numerous discussions on aspects of logic and Haskell semantics.

References

[1] Jean-Yves Girard, Yves Lafont, and P. Taylor. *Proofs and types.* Cambridge University Press, 1989.

[2] Carl A. Gunter. *Semantics of Programming Languages: Programming Techniques.* The MIT Press, Cambridge, Massachusetts, 1992.

[3] William Harrison, Timothy Sheard, and James Hook. Fine Control of Demand in Haskell. In *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany (to appear)*, 2002.

[4] Simon Peyton Jones and editors John Hughes. Report on the programming language Haskell 98. Technical Report YALEU/DCS/RR-1106, Yale University, CS Dept., February 1999.

[5] Programatica Home Page. `www.cse.ogi.edu/PacSoft/projects/programatica`. James Hook, Principal Investigator.

[6] P. Wadler. The essence of functional programming. *19th POPL*, pages 1–14, January 1992.

Applying ELAN Strategies in Simulating Processors over Simple Architectures [1]

Mauricio Ayala-Rincón [2]

Departamento de Matemática, Universidade de Brasília
Brasília D.F., Brasil

Rinaldi Maya Neto, Ricardo P. Jacobi [3]

Departamento de Ciência da Computação, Universidade de Brasília
Brasília D.F., Brasil

Carlos H. Llanos [4]

IESB, Brasília D.F., Brasil

Reiner W. Hartenstein [5]

Fachbereich Informatik, Universität Kaiserslautern
Kaiserslautern, Germany

Abstract

The simulation of processors over simple architectures is important for enabling test and verification prior to the expensive implementation involved in the development of new hardware technologies. Arvind's group has illustrated how to describe processors by term rewriting systems and has introduced a technique for proving the correctness of specifications for elaborated processors with respect to basic ones. They propose that the described processors should be simulated over standard hardware description languages such as Verilog, after translating these rewrite descriptions adequately, and not directly over the rewriting specifications. In this work we show how rewriting-logic may be applied for purely rewriting based specification as well as simulation of processors. Furthermore, we show how rewriting based simulation may be used for evaluating the performance of important hardware aspects of processors. Rewriting-logic environments such as ELAN, the one we use here, are sufficiently versatile to allow for adequate specifications and simulations which through easy modifications of the strategies enable a dynamic verification of aspects intrinsically related to hardware properties such as the size and control of reorder buffers and the method of predictions used by speculative processors.

Keywords: *Rewriting strategies and logic, hardware design and verification.*

1 Introduction

In recent years some work on applying rewriting techniques to the design of hardware has been developed. In particular, Arvind's group at MIT has treated the implementation of processors over simple architectures [12,13,1], rewrite based description and synthesis of simple logical digital circuits [7] and description of cache protocols over memory systems [16,9]. Their work has made evident the great capacity and possibilities of rewriting as an effective framework for dealing with simulation and estimation of hardware before expensive physical implementations are done. In this work we discuss the advantages and difficulties of a real rewrite based simulation of descriptions of processors over simple architectures. For this purpose we use the well-known rewriting-logic environment ELAN [5,4].

Our work, as that of Arvind group's in [12,13,1], is focused on the implementation of processors over the $\mathcal{AX}$ RISC architecture. Rules for the processors are specified in the ELAN system and different architectural components such as memory, registers, etc. are discriminated in a natural way, taking advantage of this typed language. Proving the soundness of the processors is thus reduced to proving that they simulate and are simulated by a basic processor. In our ELAN approach the separation between logic and rewriting allows us to define rules for the instructions of the processors and to specify strategies describing architectural characteristics as the size of reorder buffers - ROBs. Unlike the approach of Arvind's group, in our implementations we can simulate the execution of assembly description programs over our rewriting-specified processors; for instance, generation of the Fibonacci sequence, quick-sort, computation of the Knuth-Morris-Pratt jump function, etc., while dynamically changing strategies for estimating the most adequate form of implementing these architectural components. This is all done without translating the rewriting specifications into hardware description languages as suggested by the approach of Arvind's group. After a simulation is performed, these estimations are given by an analysis of the ELAN statistics for the number of times each rewriting rule (i.e. processor instruction) is applied. Other important architectural aspects such as predictions in processors with speculative execution are implemented in their own rewriting rules. Rewrite based simulation of programs in assembly description does not correspond exactly to the execution of these programs over real architectures, since many additional steps are executed in a rewriting based system; in particular, many unsuc-

[1] Work supported by the Brazilian/German cooperation of the CAPES/DFG foundations.
[2] Partially supported by the FEMAT Brazilian foundation. `ayala@mat.unb.br`
[3] `{rinaldi,rjacobi}@cic.unb.br`
[4] `llanos@unb.br`
[5] `hartenst@rhrk.uni-kl.de`

cessful attempts to apply rewriting rules slow down the simulation. However, ELAN statistics allow for a concrete estimation of how these processors should work in real hardware implementations.

Additionally, we point out some interesting problems inherent in the way rewriting rules (and strategies) are applied in true purely rewrite based systems. In particular, that rules are not naturally applied in a non-deterministic manner; they are selected as the first applicable rule found in the order the rules are defined and applied in the first positions (left-most, inner-most or similarly) that they match over the target term. In our implementation these problems arise when important architectural aspects as *out-of-order execution* of instruction templates over ROBs are to be simulated. Although our implementations are deterministic we comment on how one can overcome these problems in a non-purely rewriting system like ELAN, where some non-deterministic strategies are available.

2 Architecture and processors descriptions

We assume familiarity with the basic concepts of computer architecture and rewriting theory as presented respectively in [6] and [3]. Additionally, we suppose the reader familiar with rewriting-logic environments like ELAN. We briefly describe the $\mathcal{AX}$ RISC architecture and the specifications in ELAN of a basic processor over this architecture and a more elaborated one that allows for speculative execution over a reorder buffer.

2.1 The $\mathcal{AX}$ RISC architecture

$\mathcal{AX}$ is a set of RISC instructions where all memory access is done by *load* and *store* instructions and the arithmetic operations are done over the registers at the register file (*rf*). A sequence of instructions that describes a program is placed at the instruction memory (*im*). The instructions are executed in-order and after each instruction execution the contents of the program counter (*pc*) is incremented by one except for branch instructions (*Jz*).

The set of different instructions of $\mathcal{AX}$, $INST$, is described as:

$$\begin{array}{rclllll} INST & \equiv & r := \mathit{Loadc}(v) & \| & r := \mathit{Loadpc} & \| & r := \mathit{Op}(r_1, r_2) \ \| \\ & & Jz(r_1, r_2) & \| & r := \mathit{Load}(r_1) & \| & \mathit{Store}(r_1, r_2) \end{array}$$

The load-constant instruction, $r := \mathit{Loadc}(v)$, puts the constant v into the register r. The load-program-counter instruction, $r := \mathit{Loadpc}$, puts the content of the program counter into the register r. The arithmetic-operation instruction, $r := \mathit{Op}(r_1, r_2)$, performs the (abstract) arithmetic operation specified by Op on the operands specified by the registers r_1 and r_2 and puts the result into the register r. The branch instruction $Jz(r_1, r_2)$, sets the program counter to the target instruction address specified by the register r_2 when the contents of the register r_1 is zero and increments the program counter by one otherwise. The load instruction, $r := \mathit{Load}(r_1)$, loads the memory cell speci-

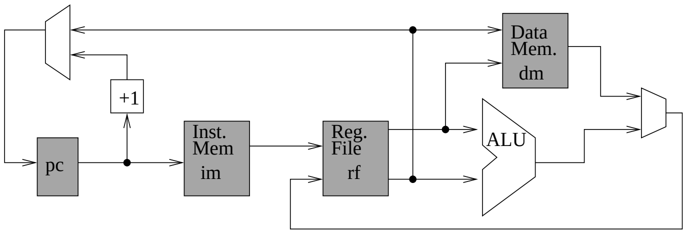


Fig. 1. Description of the basic processor

fied by the register r_1 into the register r. The store instruction, $Store(r_1, r_2)$, stores the contents of the register r_2 into the memory cell specified by the register r_1.

2.2 Basic processor

The operational semantics of the $\mathcal{AX}$ RISC instruction set is defined by a single-cycle, non pipelined, in-order execution processor that we will call the **basic processor**. See figure 1 for a description in register transfer level.

The description of the system `Sys` is given by its memory `m` and processor `Proc`: `Sys(m,Proc)`, the last that consists of the instruction address `ia` of the *pc*, the register file `rf` and the program `prog`: `Proc(ia,rf,prog)`.

The rewriting rules implementing the $\mathcal{AX}$ instructions in ELAN are given in the Table 1. The implementation follows straightforwardly from the priorly given operational semantics of these instructions. We only explain the most complex of these rewriting rules: the branch instruction *Jz*. All other rules are similarly explained.

Whenever the current instruction of the program `prog` at the position (of the instruction memory *im*) given by the instruction address `ia` is a branch instruction of the form `Jz(r1,r2)`, the program counter should be changed either by the contents of the register `r2` or by `ia+1`. The former, in the case that the contents of the register `r1` equals to zero (checked by `valueofReg(r1,rf) == 0`); the last, otherwise (checked by `valueofReg(r1,rf) != 0`). The auxiliary premise `isinstJz(selectinst(prog,ia))` checks whether the current instruction is a branch. The role of the "`where _ :=() _`" commands is to set auxiliary variables.

2.3 Implementation of a processor with speculative execution over a ROB

As in [12,13,1] more sophisticated processors may be described by rewriting rules and proved *correct* by showing that they are simulated by the basic processor and simulate the basic processor. Here we describe the implementation of a processor that does speculative execution over a ROB. See figure 2. A ROB holds instructions that have been decoded but have not completed their execution. Conceptually, the ROB divides the processor into two asynchronous parts. The first one fetches the instruction and after decoding and renaming registers, dumps it into the next available slot in the ROB. The

```
[Loadc] Sys(m,Proc(ia,rf,prog)) => Sys(m,Proc(ia+1,insertRF(rf,r,v),prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadc(instIa)
     where r :=() nameofLoadc(instIa)
     where v :=() valueofLoadc(instIa) end
[Loadpc] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r,ia),prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadpc(instIa)
     where r :=() nameofLoadpc(instIa) end
[Op] Sys(m,Proc(ia,rf,prog)) => Sys(m,Proc(ia+1,insertRF(rf,r,v),prog))
     where instIa :=() selectinst(prog,ia) if isinstOp(instIa)
     where r1 :=() reg1ofOp(instIa) where r2 :=() reg2ofOp(instIa)
     where r :=() nameofOp(instIa) where v:=() valueofOp(r1,r2,rf) end
[Jz] Sys(m,Proc(ia,rf,prog)) => Sys(m,Proc(nia,rf,prog))
     where instIa :=() selectinst(prog,ia) if isinstJz(instIa)
     where r1:=() reg1ofJz(instIa) where r2:=() reg2ofJz(instIa)
     choose try where nia:=()ia+1 if valueofReg(r1,rf)!=0
            try where nia:=()valueofReg(r2,rf) if valueofReg(r1,rf)==0
     end end
[Load] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r0,v0),prog))
     where inst :=() selectinst(prog,ia) if isinstLoad(inst)
     where r0 :=() nameofLoad(inst) where v0 :=() getMem(inst,rf,m) end
[Store] Sys(m,Proc(ia,rf,prog)) =>
 Sys(insertMEM(m,valueofReg(rA,rf),valueofReg(rB,rf)),Proc(ia+1,rf,prog))
     where inst :=() selectinst(prog,ia) if isinstStore(inst)
     where rA :=() nameofStoreR1(inst)
     where rB :=() nameofStoreR2(inst) end
```

Table 1
Rewriting rules for the basic processor

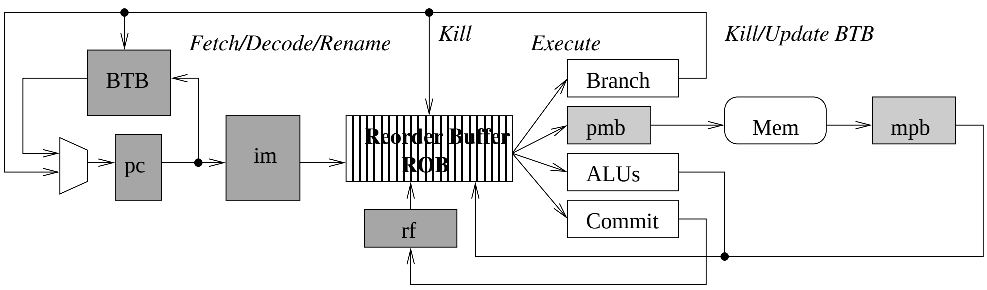


Fig. 2. Description of the speculative processor

ROB slot index serves the purpose of the renaming tag, and the instruction templates in the ROB (ITB) always contain tags or values instead of register names. An instruction template in the ROB can be solved ("executed") if all its operands are available. The second part takes any enabled instruction out of the ROB and dispatches it to an appropriate functional unit, including the memory system (then "execution" is completed). This mechanism is very similar to the execution mechanism in data flow architectures. Such an ar-

```
[PsOp]
 Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-Op(v,v1),wf,sf).itbs2,btb,prog)) =>
 Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-execOponval(v,v1),wf,sf).itbs2,
 btb,prog)) end
[PsValueForward]
 Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-v,wf,sf).itbs2,btb,prog)) =>
 Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-v,wf,sf).ValueForward(t(k),v,itbs2),
 btb,prog)) if TagExists(t(k),itbs2) end
[PsValueCommit]
 Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-v,Wreg(r),sf).itbs2,btb,prog)) =>
 Sys(m,Proc(ia, insertRF(rf,r,v),itbs2,btb,prog))
     if not TagExists(t(k),itbs2) end
```

Table 2
Arithmetic Operation and Value Propagation Rules

chitecture may execute instructions out-of-order, especially if functional units have different latencies or there are data dependencies between instructions. Additionally, speculative execution of instruction is allowed. The speculative mechanisms predicts the address of the next instruction to be issued based on the past behavior of the programs. The address of the speculative instruction is determined by consulting a table known as the branch target buffer - BTB, which can be indexed by the current content of the program counter. If the prediction turns out to be wrong, the speculative instruction and all the instructions issued thereafter are abandoned and their effect on the processor state nullified. The BTB is updated according to some prediction scheme after each execution branch resolutions.

As for the basic processor the system `Sys` is described by its memory `m` and processor `Proc`: `Sys(m,Proc)`. But in contrast, the processor consists of the `ia` of the program counter, the register file `rf`, the program `prog` as well as of the ITB (`itb`) and the BTB (`btb`): `Proc(ia,rf,itb,btb,prog)`.

In the sequel we present the ELAN implementation of the rules of the speculative processor and after that we explain the operational semantics of some of these rules. The rules are divided into four classes: arithmetic and value propagation rules; instruction issue rules; branch completion rules and memory access rules. These sets of rewriting rules are presented in the Tables 2, 3, 4 and 5, respectively. Instead of the symbol "`:=`", that is reserved in ELAN, setting in the issued rules at the ITB is denoted by "`|-`".

Arithmetic operation and value propagation rules (Table 2) deal with the computation of arithmetic operations (`PsOp`), the propagation of its results through the ITB (`PsValueForward`) and the exclusion of the instruction template from the ITB when the result had already been solved and committed to the register file (this means the renaming tag it addresses does not occur in the buffer anymore, what is decided by `TagExists(t(k),itbs)`). A value is only committed to the register file when the instruction referencing it is on the head of the ITB, this approach is conservative since it avoids the need to

```
[PsLoadcIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,
 Proc(ia+1,rf,insEndITBs(ITB(ia,k,t(k)|-v,Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadc(instIa)
     where r:=() nameofLoadc(instIa)
     where v:=() valueofLoadc(instIa)
     where k:=() lengthof(itbs)+1 end
[PsLoadpcIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,
 Proc(ia+1,rf,insEndITBs(ITB(ia,k,t(k)|-ia,Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadpc(instIa)
     where r :=() nameofLoadpc(instIa)
     where k :=() lengthof(itbs)+1 end
[PsOpIssue]Sys(m,Proc(ia,rf, itbs,btb,prog)) => Sys(m,Proc(ia+1,rf,
 insEndITBs(ITB(ia,k,t(k)|-Op(k1,k2),Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstOp(instIa)
     where r1 :=() reg1ofOp(instIa) where r2 :=() reg2ofOp(instIa)
     where r :=() nameofOp(instIa) where k :=() lengthof(itbs)+1
     where k1 :=() searchforLastTag(r1,rf,itbs)
     where k2 :=() searchforLastTag(r2,rf,itbs) end
[PsJzIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,Proc(pia,rf,
 insEndITBs(ITB(ia,k,Jz(k0,k1),NoWreg,Spec(pia)),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstJz(instIa)
     where r1 :=() reg1ofJz(instIa)
     where r2 :=() reg2ofJz(instIa)
     where k :=() lengthof(itbs)+1
     where k0 :=() searchforLastTag(r1,rf,itbs)
     where k1:=()searchforLastTag(r2,rf,itbs)
     where pia:=()getbtb(ia,btb) end
[PsLoadIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,Proc(ia+1,rf,
 insEndITBs(ITB(ia,k,t(k)|-Load(k1),Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstLoad(instIa)
     where r:=()nameofLoad(instIa)
     where r0:=()reg1ofLoad(instIa)
     where k:=()lengthof(itbs)+1
     where k1:=() searchforLastTag(r0,rf,itbs) end
[PsStoreIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,Proc(ia+1,rf,
 insEndITBs(ITB(ia,k,Store(k0,k1),NoWreg,NoSpec),itbs),btb,prog))
     where instIa:=()selectinst(prog,ia) if isinstStore(instIa)
     where r0 :=() nameofStoreR1(instIa)
     where r1:=()nameofStoreR2(instIa)
     where k:=()lengthof(itbs)+1
     where k0:=()searchforLastTag(r0,rf,itbs)
     where k1:=()searchforLastTag(r1,rf,itbs) end
```

Table 3
Instruction Issue Rules

reconstruct the state of the register file in the event of wrong speculations.

Issue rules (Table 3) are those used for the issuing of the instructions which generate templates stored in the ITB. Branch completion rules (Table 4) are those which deal with the resolution of speculations. When a branch instruc-

[PsJumpCorrectSpec]
```
 Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(0,nia),wf,Spec(pia)).itbs,btb,prog)) =>
 Sys(m,Proc(ia,rf,itbs,btb,prog)) if pia==nia end
```
[PsJumpWrongSpec]
```
 Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(0,nia),wf,Spec(pia)).itbs,btb,prog)) =>
 Sys(m,Proc(nia,rf,nilitb,btb1,prog))
      if pia!=nia where btb1:=()changebtb(ia1,nia,btb) end
```
[PsNoJumpCorrectSpec]
```
 Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(v,nia),wf,Spec(pia)).itbs,btb,prog)) =>
 Sys(m,Proc(ia,rf,itbs,btb,prog)) if v != 0 and pia == ia1+1 end
```
[PsNoJumpWrongSpec]
```
 Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(v,nia),wf,Spec(pia)).itbs,btb,prog)) =>
 Sys(m,Proc(ia1+1,rf,nilitb,btb1,prog))
      if v != 0 and pia != ia1+1
      where btb1 :=() changebtb(ia1,ia1+1,btb) end
```

Table 4
Branch Completion Rules

[PsLoad]`Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-Load(v),wf,sf).itbs,btb,prog))`
```
 => Sys(m, Proc(ia,rf,ITB(ia1,k,t(k)|-v0,wf,sf).itbs,btb,prog))
      where v0 :=() valueofMem(v,m) end
```
[PsStore]`Sys(m,Proc(ia,rf,ITB(ia1,k,Store(a,v),wf,sf).itbs,btb,prog))`
```
 => Sys(insertMEM(m,a,v), Proc(ia,rf,itbs,btb,prog)) end
```

Table 5
Memory Access Rules

tion is issued the processor has to know which will be the next instruction to be fetched. The next predicted instruction is indicated by the BTB that is an indexed table. Suppose that the program instruction at position `ia` is being issued, the next value of the program counter, called `pia`, is looked up in the BTB using as index the current program counter (`pia :=() getbtb(ia,pia)`) and then the execution resumes at the `pia` value. When the ITB element containing the branch instruction reaches the head of the ITB it is the time to check if the speculation was done correctly or if the processor needs to fix the mistake and restart the execution at the correct program counter value. In the last case, the remaining instructions already in the ITB should be ignored. The rules in the Table 4 deal with this issue. Exemplifying, suppose that the head of the ITB is of the form `ITB(ia,k,Jz(v,nia),wf,Spec(pia))`, the branch completion rule has to check whether the value `v` is zero or not and then, respectively, check whether either the speculated address `pia` coincides with `nia` or with `ia+1`. In this event the prediction has been proved correct and the execution resumes. Otherwise the program counter must be set, respectively, either to the value of `ia+1` or to the correct value of the branch represented by `nia`, depending on whether the wrong speculation was a no jump or a jump, and the ITB must be completely emptied because the remaining instructions should not be executed. These rules also control the updating of the BTB for dynamic speculation (through the rules which define `changebtb`).

The memory access rules (Table 5) `PsLoad` and `PsStore` deal with the communication between the ROB and the data memory `m`. These rules are applied after the processor has resolved all values of the tags of the instruction templates stored in the ITB by the `PsLoadIssue` and `PsStoreIssue` issue rules, respectively.

2.4 Proving correctness of processors

One useful feature of rewrite based specification of processors is the possibility of proving the correctness of the implementation of some instruction set describing a processor. This is done by showing that one implementation simulates another in regard of some observation function. The main idea, according to [12,13,1] is to design a function that can extract all the programmer visible states; i.e., the program counter, the register file and the memory from the system. The proof of the correctness of our specification of the speculative processor, here given for completeness of the presentation, follows the lines in [12].

It is easy to show that the speculative processor simulates the basic one. One basic processor term can be "upgraded" to one of the speculative processor simply by adding an empty ITB and an arbitrary BTB to the processor.

Definition 1 (ITBL) *The Instruction Template Buffer Lift of a basic processor term is defined by*

$$ITBL(\texttt{Sys(m,Proc(ia,rf,prog))}) \equiv \texttt{Sys(m,Proc(ia,rf,nilitb,btb,prog))}$$

where `btb` *is an arbitrary BTB and* `nilitb` *and empty ITB.*

Theorem 1 *Let s and t be system terms of the basic processor. If $s \to^* t$ in the basic processor, then $ITBL(s) \to^* ITBL(t)$ in the speculative processor.*

Proof. Sequences of rules of the speculative processor can simulate each basic processor rule. For example, the `Op` rule in the basic processor can be simulated by consecutively applying the `PsOpIssue`, `PsOp` and `PsValueCommit` rules in the speculative processor; the `Load` rule in the basic processor can be simulated by consecutively applying the `PsLoadIssue`,`PsValueForward`, `PsLoad` and `PsValueCommit` rules in the speculative processor; etc. □

Now we need to define a projection function from the speculative processor to the basic processor. This is not simple because of the partially executed instructions. The approach in [12] is based on flushing instructions in the ITB. The key observation is that during some time of execution over an speculative processor, if no instruction is issued then the ITB will soon become empty. Only instruction issue rules can further expand the ITB. So, we can define another rewriting system which uses the same grammar as the speculative processor and include all its rules except the instruction issue ones.

Definition 2 *The rewriting system R_{ITBF} over terms of the speculative processor is given by the set of rewriting rules* {`PsOp`, `PsValueForward`, `PsValueCommit`, `PsJumpCorrectSpec`, `PsJumpWrongSpec`, `PsNoJumpCorrectSpec`, `PsNoJumpWrongSpec`, `PsLoad`, `PsStore`}.

One can prove that the rewriting system R_{ITBF} is strongly terminating and confluent and that its normal forms have always empty ITBs.

Definition 3 (ITBF) *Let* `Sys(m,Proc(ia,rf,nilitb,btb,prog))` *be the R_{ITBF} normal form of a given term of the speculative processor s. The* instruction template buffer flush *of s, denoted by ITBF(s), is the result of deleting from this R_{ITBF} normal form its empty ITB and its BTB:* `Sys(m,Proc(ia,rf,prog))`.

Theorem 2 *Let s and t be system terms of the speculative processor. If $s \to^* t$, then ITBF(s) $\to^*$ ITBF(t) in the basic processor.*

Speculative processor issue rules	basic processor rules
`PsLoadcIssue`	`Loadc`
`PsLoadpcIssue`	`Loadpc`
`PsOpIssue`	`Op`
`PsJzIssue`	`Jz`
`PsLoadIssue`	`Load`
`PsStoreIssue`	`Store`

Table 6
Correspondence between speculative issue rules and basic processor rules

Proof. The proof is by induction on the number of rewrite steps n on the derivation $s \to^n t$. For $n = 0$ this is obvious. For the inductive step, assume $s \to t$ by applying the rule α. If $\alpha \in R_{ITBF}$, then *ITBF*(s) and *ITBF*(t) coincide. If $\alpha \notin R_{ITBF}$, that is α is an instruction issue rule, then we will prove that either *ITBF*(s) and *ITBF*(t) coincide or *ITBF*(s) can be rewritten into *ITBF*(t) by applying an appropriate basic processor rule.

Suppose $s \to s_1$ by applying a rule $\beta \in R_{ITBF}$. We have two cases to consider:

Case 1. β is a mis-prediction-recover rule: `PsJumpWrongSpec` or `PsNoJump-WrongSpec`. By applying β to t we have $t \to s_1$, since the instruction issuing will be canceled by the mis-prediction-recover rule.

Case 2. β is not a mis-prediction-recover rule. In this case we can notice that α can also be applied to s_1. Suppose that $s_1 \to t_1$ by applying α. If α is `PsValueCommit` and the register to which the value is committed is referenced as an operand register in the instruction issued by α, then $t \to^* t_1$ by first applying once or twice `PsValueForward`, and then applying the rule β. Otherwise $t \to t_1$ by applying β.

Let s_n be the R_{ITBF} normal form of s. We have two cases to consider:

Case 1. If this normalization from s into s_n invokes one mis-prediction-recover rule, then there are terms s_i, t_i and s_{i+1} such that $s_i \to t_i$, by applying α, $s_i \to s_{i+1}$ and $t_i \to s_{i+1}$ by applying the mis-prediction-recover rule. This implies that s and t have identical R_{ITBF} normal forms.

Case 2. In the other case, by induction, we have that α can be applied to s_n

to yield t_n such that $t \to^* t_n$ by applying just R_{ITBF} rules.

Let t_{n+1} be the R_{ITBF} normal form of t_n. Since s_n and t_{n+1} both have an empty ITB, we can easily show that $ITBF(s_n) \to ITBF(t_{n+1})$ by applying the corresponding basic processor rule according to the Table 6. $\square$

3 Benefits of the separation between logic and rewriting in simulating processors

The natural separation in ELAN between rewriting and logic enables the controlled application of rules (i.e., processor instructions) and the adequate simulation of many interesting elements of hardware. For instance, the size of ROBs is one of the basic hardware ingredients of the speculative processor that is controlled by ELAN strategies. In fact, ROBs are controlled by specifying strategies which restrict the number of applications of issue rules. Suppose you want to simulate a ROB of size n, that should completely be filled and emptied alternately. Then the following simple ELAN strategy is used:

$$\mathit{repeat}* \left(\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules}); \\ \quad \left.\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\ \qquad\vdots \\ \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \end{array}\right\} n\text{-}1 \\ \mathit{normalise}(\mathit{first\ one}(\mathit{non\_issue\_rules})) \end{array} \right)$$

Other strategies for handling the ROBs can similarly be specified. For example, for maintaining a ROB of size n filled during the whole execution, one can start as before, but in the subsequent normalization with all non issue rules (R_{ITBF} normalization) these rules should be treated individually. This treatment depends on whether the given non issue rule maintains or decreases the number of instruction templates in the ROB. For instance, since after a wrong branch speculation (rule `PsJumpWrongSpec`) the ROB is emptied the strategy should immediately fill the ROB by applying n issue rules. Below we sketch this strategy showing the case of the rule `PsJumpWrongSpec`.

$$\mathit{repeat}* \left(\begin{array}{l} \left.\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules}); \\ \quad \left.\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\ \qquad\vdots \\ \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \end{array}\right\} n\text{-}1 \end{array}\right\} \text{Initialization filling the ROB} \\ \mathit{normalise} \left(\mathit{first\ one} \left(\begin{array}{r} \left.\vdots\ \right\} \begin{array}{l}\text{treatment of all}\\ \text{other non issue rules}\end{array} \\ \left[\begin{array}{l} \mathtt{PsJumpWrongSpec}; \\ \mathit{first\ one}(\mathit{issue\_rules}); \\ \quad \left.\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\ \qquad\vdots \\ \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \end{array}\right\} n\text{-}1 \end{array} \right] \\ \left.\vdots\ \right\} \begin{array}{l}\text{treatment of all}\\ \text{other non issue rules}\end{array} \end{array} \right) \right) \end{array} \right)$$

In contrast to the control of ROBs other interesting aspects of processors as the method of branching prediction are directly controlled by the rewriting rules. The advantages of having ROBs is that instruction templates may be charged and these templates partially executed by the pipeline control. When at a point of the computation, determined by the program counter `ia`, a branch instruction template `Jz(r1,r2)` is charged into the ROB, it is undecided which is the following instruction template to be charged into the ROB, since at this point of the computation the values of the tags associated with the registers `r1` and `r2` are not necessarily resolved. Thus in speculative processors one has to decide which instruction template is the next to be charged according to the contents for the `ia` in the BTB. Well-known dynamic branch prediction schemes are specified by simple rewriting rules. We mention here the *1-bit* and *2-bit dynamic prediction* methods [14]. Initially, any prediction is given in the BTB. For instance, one can give pairs $(1,2),\ldots,(j,j+1),\ldots,(n,n+1))$ meaning that after execution of the j^{th} instruction the prediction is to jump to the next instruction $(j+1^{th})$ of the program. These predictions (i.e., pairs) are only necessary for the addresses of branch instructions in the program. Subsequently, the predictions are modified according to the execution history.

In 1-bit dynamic prediction, the prediction for the n^{th} instruction is actualized according to the next instruction to be executed. Once a prediction fails the corresponding value in the BTB is changed to the correct address of the instruction to be executed.

In 2-bit dynamic prediction, there are four different states of the prediction: *strongly taken, weakly taken, weakly not taken, strongly not taken.* If the state is either *strongly (not) taken* or *weakly (not) taken* and the prediction is correct: "jump" ("next instruction"), then the state is changed to *strongly (not) taken.* If the state is *strongly (not) taken* and the prediction fails: "next instruction" ("jump"), then the state is changed to *weakly (not) taken.* If the state is *weakly (not) taken* and the prediction fails: "next instruction" ("jump"), then the BTB is modified according to the correct address given by the contents of the second register of the branch instruction and the state is changed to *weakly not taken* (*weakly taken*).

Both prediction strategies are specified and simulated by purely rewriting. This is implemented by simple boolean conditions over the branch completion rules: comparisons between `pia` (predicted instruction address), `nia` (next correct instruction address) and `ia+1` (next instruction address in the program) for the four branch completion rules in the Table 4. Once a prediction fails, the BTB is modified by the function `changebtb`, that is specified by purely rewriting and adapted for the two prediction methods.

Furthermore, the performance of different ways to implement proposed processors can be determined by analyzing the ELAN statistics. For instance, one can estimate whether 1-bit performs better than 2-bit prediction for the execution of an assembly description of quick-sort over the speculative processor implemented with the strategy of alternatively filling and emptying the ROB. The total number of wrong and correct predictions (i.e., number of applications of branch completion rules in the Table 4) for ordered (the worst-case of quick-sort) and random lists are given in the Table 7. The observation of the differences between the number of wrong predictions for both methods gives an important insight about the advantages of 2-bit over 1-bit

	Size	10 ran	10 ord	20 ran	20 ord	30 ran	30 ord	40 ran	40 ord	50 ran	50 ord
1-bit	correct	30	60	109	225	185	490	278	855	401	1320
	wrong	34	34	72	74	114	114	159	154	196	194
2-bit	correct	28	73	120	258	194	543	286	928	407	1413
	wrong	36	21	61	41	105	61	151	81	200	101

Table 7

Elan statistics for quick-sort executed with 1-bit and 2-bit dynamic predictions

prediction, since in the worst-case a wrong prediction flushes the ROB which has been filled with instruction templates over which previous operations have been executed. One can check on the table that the differences between the number of wrong predictions for the two methods is much more significant for ordered lists than for random lists. Consequently, the physical hardware implementation of a processor dedicated to this kind of sorting for random inputs can be performed with the simplest (and cheaper) 1-bit method.

Important aspects like out-of-order execution are not easy to implement in a natural way in practical purely rewrite based programming environments. In fact, out-of-order execution of instruction templates over a ROB can only be simulated by allowing a truly non-deterministic application of the rewriting rules (i.e., processor instructions) over the ROB during any time of the computation. For allowing out-of-order execution, instead of the usual CONS operator "." of instruction templates and ITBs (which appears as `inst_temp.itbs` in our implementation) a new operator "#" is defined for concatenating ITBs and/or instruction templates. Thus ITBs are represented as `itbs1#inst_temp#itbs2` being `itbs1` and `itbs2` lists of instruction templates and `inst_temp` a sole instruction template. Rules of the rewriting system are modified by replacing all their ITBs with this new representation as we illustrate for the `[PsOp]` rule below:

```
Sys(m,Proc(ia,rf,itbs1#ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)#itbs2,btb,prog)) =>
Sys(m,Proc(ia,rf,itbs1#ITB(ia1,k,t(k)|-execOponval(v,v1),wf,sf)#itbs2,btb,
                                                              prog)) end
```

By matching the instruction template, `ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)`, the new `[PsOp]` rule can be applied not only at the first but at any position of the current ITB: `itbs1#ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)#itbs2`. In the theory, the rewriting system obtained by modifying all the rules as suggested above enables out-of-order execution, since rewriting rules are applied non-deterministically. But in the practice, in purely rewrite based programming environments, this solution does not work since the application of a rule is decided by searching for either left-most or right-most (inner-most) redices over the ITBs (according to the way the constructor "#" is defined) [8].

For rewriting based implementations of a real out-of-order execution mechanism, the availability of true non-deterministic strategies is necessary. With some additional effort, in a rewriting-logic based system as ELAN strategy constructors like *don't know choose* (that gives all possible reducts) can be adapted for simulating the needed non-determinism over the ROBs [15,10,11].

4 Conclusions and future work

We have shown how processors may be specified and their execution simulated over rewriting(-logic) systems. Unlike Arvind's group, who proposes the simulation of the execution of these specifications over standard hardware description languages, we address the simulation of the execution of processors directly over the rewriting specification avoiding the cost of program translation. Furthermore, we have illustrated why the rewriting part as well as the logical part of ELAN are adequate for the simulation of simple hardware components like the method of prediction in speculative processors (done in our case by pure rewriting) and control of the size of ROBs (done in our case by logic strategies). After having specified the rewriting rules for the instruction set of a processor, the intrinsic separation between logic and rewriting in ELAN results in enough versatility for dealing with different conceptions of manipulation of ROBs without additional effort in these rewrite specifications. Additionally, we illustrate how statistics of the application of rewriting rules may be used for estimating and comparing the performance of different processors. Although not done in our implementation, non-deterministic strategies implemented in ELAN also may be shown to be adequate for simulating essential hardware conceptions of these processors like the *out-of-order* execution of instruction templates in ROBs.

Through rewriting-logic one can describe an architecture as precisely as one wants. For example, rules of the speculative processor may be atomized in order to reflect the behavior of lower-level hardware components such as pipelines and functional units of processors like fetch, decode and execution units. In fact, one of our current goals is to analyze the possibilities of using rewriting for synthesizing (logic components for building) logical circuits for arithmetic operators at their layout level [2]. One of the interesting aspects that emerges from this study is the necessity of new hardware oriented notions of normal forms, since the more adequate algebraic expressions to be transformed into circuits are the ones with *more regularities*. These are consequently the ones that can be implemented with the smallest number of different classes of atomic hardware components, and are not the simplest ones from the algebraic point of view, which is the norm in rewriting.

References

[1] Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors. Technical Report 419, Laboratory for Computer Science - MIT, 1999. Also in IEEE Micro Special Issue on "Modeling and Validation of Microprocessors", 1999.

[2] M. Ayala-Rincón, R. W. Hartenstein, R.P. Jacobi, and C. Llanos. Designing Arithmetic Digital Circuits via Rewriting-Logic. Available at `www.mat.unb.br/~ayala/publications.html`, 2002.

[3] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Science Publishers, 1998.

[5] H. Cirstea and C. Kirchner. Combining Higher-Order and First-Order Computation Using ρ-Calculus: Towards a Semantics of ELAN. In *Frontiers of Combining Systems 2*, Studies on Logic and Computation, 7, chapter 6, pages 95–121. Research Studies Press/Wiley, 1999.

[6] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, second edition, 2002.

[7] J. C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. Technical Report 421 A, Laboratory for Computer Science - MIT, 1999. Also in Proc. of the Tenth IFIP International Conference on VLSI - VLSI 1999.

[8] H. Hussmann. *Nondeterminism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, 1993.

[9] Arvind J. Stoy, X. Shen. Proofs of Correctness of Cache-Coherence Protocols. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 43–71, 2001.

[10] H. Kirchner and P.-E. Moreau. Non-deterministic computations in ELAN. In J.L. Fiadeiro, editor, *Recent Developements in Algebraic Specification Techniques, Proc. 13th WADT'98, Selected Papers*, volume 1589 of *LNCS*, pages 168–182, 1998.

[11] H. Kirchner and P.-E. Moreau. Promoting Rewriting to a Programming Language: A Compiler for Non-Deterministic Rewrite Programs in Associative-Commutative Theories. *J. of Functional Programming*, 11(2):207–251, 2001.

[12] X. Shen and Arvind. Design and Verification of Speculative Processors. Technical Report 400 A, Laboratory for Computer Science - MIT, 1998. Also in Proc. of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden, 1998.

[13] X. Shen and Arvind. Modeling and Verification of ISA Implementations. Technical Report 400 B, Laboratory for Computer Science - MIT, 1998. Also in Proc. of the Australasian Computer Architecture Conference, 1998.

[14] D. Sima, T. Fountain, and P. Kacsuck. *Advanced Computer Architectures: a Design Space Approach.* Addison-Wesley, 1997.

[15] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. In H. Ganzinger, editor, *Proc. Seventh Int. Conf. on Rewriting Techniques and Applications*, volume 1103 of *LNCS*, pages 154–168, 1996.

[16] L. Rudolph X. Shen, Arvind. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. In *International Conference on Supercomputing*, pages 135–144. ACM, 1999.

Author Index

Albert, Elvira 97
Ayala-Rincón, Mauricio 127
Brand, Mark G.J. van den 21
Glauert, John 81
Hanus, Michael 97
Hartenstein, Reiner 127
Huch, Frank, ... 97
Jacobi, Ricardo P. 127
Khasidashvili, Zurab 81
Klint, Paul 21
Lämmel, Ralf 39
Llanos, Carlos Llanos 127
Middeldorp, Aart 1
Neto, Rinaldi Maya 127
Oliver, Javier 97
Olmos, Karina 59
Oostrom, Vincent van 79
Réty, Pierre 3
Vidal, Gérman 97
Vinju, Jurgen J. 21
Visser, Eelco 59
Vuotto, Julie 3