Zoltan Esik
University of Szeged, Hungary
Igor Walukiewicz
Bordeaux University, France

ETAPS 2003 April 5-13, Warsaw, Poland

FICS'03 - FIXED POINTS IN COMPUTER SCIENCE

EUROPEAN JOINT CONFERENCES
ON THEORY AND PRACTICE
OF SOFTWARE 2003

APRIL 5–13, WARSAW, POLAND

Fixed Points in Computer Science

Proceedings of an International Workshop

EDITORS:
ZOLTAN ESIK
IGOR WALUKIEWICZ

Warszawa, 2003

Druk i oprawa: Zakład Graficzny UW, zam. 248/2003

Table of Contents

Primitive Recursion for Rank-2 Inductive Types

Andreas Abel[*] and Ralph Matthes[**]

Department of Computer Science
University of Munich

Recently, higher-rank datatypes have drawn interest in the functional programming community [Oka99,Oka96,Hin01]. Rank-2 non-regular types, so-called *nested datatypes*, have been investigated in the context of Haskell. To define total functions which traverse nested datastructures, Bird et al. [BP99] have developed *generalized folds* which implement an iteration scheme and are strong enough to encode most of the known algorithms for nested datatypes. In this note, we investigate a scheme to overcome some limitations of iteration which we expound in the following.

Since the work of Böhm *et al.* [BB85] it is well-known that iteration for rank-1 datatypes can be simulated in typed lambda-calculi. The easiest examples are iterative definitions of addition and multiplication for Church numerals. The iterative definition of the predecessor, however, is inefficient: It traverses the whole numeral in order to remove one constructor. Surely, taking the predecessor should run in constant time.

Primitive recursion is the combination of iteration and efficient predecessor. A typical example for a prim. rec. algorithm is the natural definition of the factorial function. It is common belief that prim. rec. cannot be reduced to iteration in a computationally faithful manner. This is because no encoding of natural numbers in the polymorphic lambda-calculus (System F) seems possible which supports a constant-time predecessor operation (see Spławski and Urzyczyn [SU99]). Mendler extended System F by a scheme of prim. rec. for rank-1 datatypes and proved strong normalization [Men87]. Mendler's formulation does not follow the usual category-theoretic approach with initial recursive algebras (see Geuvers [Geu92]).

For rank-2 datatypes there are also examples of functions which can most naturally be implemented with prim. rec. One is *redecoration for triangular matrices* which is presented below. These examples are not instances of generalized folds à la Bird *et al.*, which remain within the realm of iteration but hardwire Kan extensions into the recursion scheme. Rank-2 prim. rec., which we propose in this work, seeks to extend rank-2 iteration in the same way that prim. rec. extends rank-1 iteration. We achieve this by lifting Mendler's scheme of prim. rec. to rank 2. The decision for Mendler-style and against the traditional way roots in the following observation: Experiments with formulations according to the traditional style showed unnecessary but unavoidable traversals of the whole data structures in our examples. Mendler's style, however, yielded precisely the

[*] The first author is supported by the *Graduiertenkolleg "Logik in der Informatik"* of the *Deutsche Forschungsgemeinschaft.*

[**] Both authors acknowledge financial support by ETAPS 2003.

desired efficient reduction behavior. This was crucial since the only reason to incorporate prim. rec. is operational efficiency as opposed to denotational expressiveness.

We work within the framework System F^ω of higher-order parametric polymorphism formulated in Curry-style, i.e., as a type assignment system for the pure lambda-calculus. For type transformers $X, Y : * \to *$ we abbreviate the type of natural transformations $\forall A.\, X\,A \to Y\,A$ from X to Y by $X \subseteq Y$. Let $\mathsf{id} = \lambda x.x$ denote the identity function.

We extend the framework by a new constructor constant μ and two term constants in and MRec and a new reduction rule as follows.

Formation.	μ	$: ((* \to *) \to * \to *) \to * \to *$
Introduction.	in	$: \forall F^{(* \to *) \to * \to *}.\, F\,(\mu F) \subseteq \mu F$
Elimination.	MRec	$: \forall F^{(* \to *) \to * \to *} \forall G^{* \to *}.\, (\forall X^{* \to *}.\, X \subseteq \mu F \to$ $X \subseteq G \to F\,X \subseteq G) \to \mu F \subseteq G$
Reduction.		$\mathsf{MRec}\,s\,(\mathsf{in}\,t) \longrightarrow_\beta s\ \mathsf{id}\,(\mathsf{MRec}\,s)\,t$

The type transfomer $\mu F : * \to *$ is the least fixed-point of the constructor $F : (* \to *) \to * \to *$ and denotes a simultaneously defined family of types of well-founded trees, their shape depending on F. For instance, using $F = \lambda X \lambda A.\, 1 + A \times X\,A$ the well-known type of polymorphic lists is recovered. The term in is the general constructor, which, in case of lists, codes together nil and cons. The term MRec establishes a scheme of primitive recursion in the style of Mendler. Typical for this style is the universally quantified constructor variable X in the type of the step term s which ensures termination without any positivity restrictions on F. During reduction, X is instantiated by μF, and the first parameter, $i : X \subseteq \mu F$, by id. The presence of a transformation i from the blank type X back into the fixed-point μF is what distinguishes Mendler-style prim. rec. from Mendler-style iteration.

$$\begin{array}{r}
A\,E\,E\,E \ldots E \\
A\,E\,E \ldots E \\
A\,E \ldots E \\
A \ldots E \\
\ddots\ E \\
A
\end{array}$$

An example of a non-regular datatype is $\mathsf{Tri}\,A = (\mu\,\mathsf{TriF})\,A$ with $\mathsf{TriF} = \lambda X \lambda A. A \times (1 + X(E \times A))$, the type of triangular matrices over a given entry type E but with type A on the diagonal. For these matrices, we define a redecoration operation

$$\mathsf{redec} : \forall A \forall B.\ \mathsf{Tri}\,A \to (\mathsf{Tri}\,A \to B) \to \mathsf{Tri}\,B.$$

The call $\mathsf{redec}\ t\,f$ replaces each diagonal element a of t with the result of applying f to the sub-triangle whose upper-left corner is a. Redecoration is a natural example for primitive recursion and is no instance of a generalized fold.

System F^ω, extended by Mendler-style primitive recursion, is still confluent and strongly normalizing. A dual construction can be carried out to obtain coinductive families with primitive corecursion.

Acknowledgement. We thank Tarmo Uustalu for communicating the example of triangular matrices to us.

References

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BP99] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In Bengt Nordström, Kent Pettersson, and Gordon Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 193–217, 1992. Electronically available via `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.dvi.Z`.

[Hin01] Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming*, 11(5):493–524, 2001.

[Men87] Nax P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y.*, pages 30–36. IEEE Computer Society Press, 1987.

[Oka96] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, Carnegie Mellon University, 1996.

[Oka99] Chris Okasaki. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *International Conference on Functional Programming*, pages 28–35, September 1999.

[SU99] Zdzisław Spławski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. *SIGPLAN Notices*, 34(9):102–113, 1999. Proceedings of the 1999 International Conference on Functional Programming (ICFP), Paris, France.

On ambiguous classes in the μ-calculus hierarchy of tree languages

A. Arnold and L. Santocanale

LaBRI, Université Bordeaux I and CNRS (UMR 5800)

Abstract. Rabin has proved that if both a set of trees and its complement are Büchi definable in the monadic second order logic then these sets are weakly definable. In terms of μ-calculus, this theorem reads as $\Pi_2 \cap \Sigma_2 = comp(\Pi_1, \Sigma_1)$.
It is natural to ask whether the equality $\Pi_n \cap \Sigma_n = comp(\Pi_{n-1}, \Sigma_{n-1})$ still holds for higher levels of the hierarchy. In this paper we prove that it is NOT the case.
We also show that Rabin's result can be generalized as follows, taking into account that any Büchi definable set is recognized by a nondeterministic Büchi automaton: If a language and its complement are recognized by nondeterministic Π_n automata then they are in $comp(\Pi_{n-1}, \Sigma_{n-1})$.

1 Introduction

It has been proved by Rabin [11] that if a tree language is both Büchi and co-Büchi then it is weakly definable. In terms of μ-calculus, a Büchi set is a set recognized by a nondeterministic automaton in Π_2, while a weakly definable set is a set recognizable by an alternating automaton in $comp(\Pi_1, \Sigma_1)$ [10, 9].

Since a set is in Π_2 if and only if its complement is in Σ_2 and since every set in Π_2 is a Büchi definable set [2, 8, 4], Rabin's result can be stated as: $\Pi_2 \cap \Sigma_2 = comp(\Pi_1, \Sigma_1)$ [3].

Therefore a natural question is whether the equality $\Pi_n \cap \Sigma_n = comp(\Pi_{n-1}, \Sigma_{n-1})$ still holds for $n > 2$.

In [5], this equality has been proved for the μ-calculus over non-distributive lattices. In this paper, we prove that it does not hold for the μ-calculus of tree languages (this implies that it does not hold for the modal μ-calculus). The proof of this result is quite similar to the proof of the strictness of the μ-calculus hierarchy [1, 4] and uses the same diagonalization argument.

But then a new question arises: why this equality is true for $n = 2$? What is the specific property of Π_2 (or Σ_2) which makes the property to hold? Indeed Π_2 is the only class (with Π_1 and Σ_1) which has the property that a term is equivalent to a "disjunctive" term in the same class [4]. [1]

Therefore another possible generalization of $\Pi_2 \cap \Sigma_2 = comp(\Pi_1, \Sigma_1)$ is: if L and its complement are recognized by nondeterministic Π_n automata, they are in $comp(\Pi_{n-1}, \Sigma_{n-1})$. Indeed, we show a stronger separation result: If $\mathcal{A}$ and $\mathcal{B}$ are two nondeterministic automata in Π_n such that $L(\mathcal{A}) \cap L(\mathcal{B}) = \emptyset$ then there is an alternating automaton $\mathcal{C}$ in $comp(\Pi_{n-1}, \Sigma_{n-1})$ such that $L(\mathcal{A}) \subseteq L(\mathcal{C}) \subseteq \overline{L(\mathcal{B})}$. We prove this result in case of (binary) tree automata. The same proof also works for modal automata introduced in [7]. It combines a technique used in [5] with the construction of an alternating automata of [9].

In the next section of the paper we introduce some definitions. The definition of an automaton that we give is not the most general one, but every automaton can

[1] "Disjunctivity" is a notion introduced in [7] to generalize the notion of nondeterminism.

be transformed into an equivalent one having this restricted form, without affecting its position in the alternating depth hierarchy.

Section 3 contains the definition of a language which is in $\Pi_n \cap \Sigma_n$ and not in $comp(\Pi_{n-1}, \Sigma_{n-1})$. The proofs of this section are not more difficult than the proof of the strictness of the μ-calculus hierarchy. Thus, they are only sketched.

Section 4 contains the proof of the separation theorem. This proof is more detailed and also self-contained, excepted for the proposition characterizing the emptyness of the intersection of two automata, which is also almost obvious. Then we show how to extend this proof to the modal μ-calculus.

2 Preliminary definitions

2.1 Tree automata

For simplicity, we consider only the case of binary trees. Let F be a set of binary symbols, and recall that an F-tree is a mapping t from $\{l,r\}^*$ to F.

An alternating tree automaton is a tuple $\mathcal{A} = \langle X, \Delta, \rho \rangle$ where:

- X is a finite set of states (note that there are not initial states).
- For each $x \in X$ and each $f \in F$, $\Delta(x,f)$ is a positive boolean combination of elements of $X \times \{l,r\}$. By using the distibutivity laws and grouping together pairs with the same direction, $\Delta(x,f)$ can also be written as a set of *rules*, where each rule r is a pair (X'_r, X''_r) of subsets of X.
- ρ is a mapping from X to $\mathbb{N}$.

An automaton is *nondeterministic* if for each rule r the sets X'_r and X''_r are singletons.

An automaton is in Π_n (resp. Σ_n) if there is an even (resp. odd) integer $m \geq n-1$ such that $\rho(X) \subseteq \{m-n+1, \ldots, m\}$.

An automaton is in $\Gamma_n = comp(\Pi_n, \Sigma_n)$ if there is a *preorder* $\succeq$ on X such that:

- for any x and f, for any rule $r = (X'_r, X''_r) \in \Delta(x,f)$ and any $x' \in X'_r \cup X''_r$, $x \succeq x'$,
- for any equivalence class X' of X induced by $\succeq$ (x is equivalent to x' if $x \succeq x'$ and $x' \succeq x$), there exists $m \geq n-1$ such that $\rho(X') \subseteq \{m-n+1, \ldots, m\}$ or $\rho(X') \subseteq \{m-n, \ldots, m-1\}$.

2.2 Tree languages

Let $\mathcal{A}$ be an automaton and t be a tree. We define the parity game $G(\mathcal{A}, t)$ as follows.

- Eva's positions are the pairs (x,u) with $x \in X$ and $u \in \{l,r\}^*$. The rank of (x,u) is $\rho(x)$.
- Adam's positions are all the pairs (r,u) where r is a rule. The rank of an Adam's position is always 0.
- There is an Eva's move from (x,u) to (r,u) if and only if $r \in \Delta(x, t(u))$.
- If $r = (X', X'')$ there is an Adam's move from (r,u) to (x', ul) for any $x' \in X'$ and to (x'', ur) for any $x'' \in X''$.

We say that t is recognized by $\mathcal{A}$ from state x if the position (x, ε) is winning. We denote by $L_x(\mathcal{A})$ the set of trees recognized by $\mathcal{A}$ from x.

We say that a tree language is in Π_n (resp. Σ_n, Γ_n) if there is an automaton $\mathcal{A}$ in Π_n (resp. Σ_n, Γ_n) and a state x such that $L = L_x(\mathcal{A})$.

The following results can be proved using the notion of a dual automaton (see [4]).

Proposition 1 *L is in Π_n if and only if $\overline{L}$ is in Σ_n.*
L is in Γ_n if and only if $\overline{L}$ is in Γ_n.

It is also known that

Proposition 2 *If $L \in \Gamma_n$ then $L \in \Pi_{n+1} \cap \Sigma_{n+1}$.*

3 The inequality theorem

We are going to show:

Theorem 3 *For any $n > 2$ there is a tree language in $\Pi_n \cap \Sigma_n$ which is not in Γ_{n-1}.*

3.1 Some tree languages

Let $n > 2$ and let A_n be the set of binary symbols $\{c_i, d_i \mid 1 \leq i \leq n\}$.

Let K_n be the set of all trees over A_n such that on each branch the set of symbols which occur infinitely often is included in $\{c_i, d_i \mid 1 \leq i \leq n-1\}$ or in $\{c_i, d_i \mid 2 \leq i \leq n\}$. This set is in Σ_2 since its complement $\overline{K_n}$ is in Π_2, because the condition that a tree has at least one branch belonging to a given regular ω-language is a Büchi condition.

Let $\mathcal{W}_n$ be the (nondeterministic) automaton (in Π_n if n is even and in Σ_n otherwise) whose set of states is $\{q_i \mid 1 \leq i \leq n\} \cup \{q_\top\}$, where the rank of q_i is i and the rank of $q_\top$ is 2, and whose transition function Δ is defined as follows:

- for any i, $\Delta(q_\top, c_i) = \Delta(q_\top, d_i) = \{(q_\top, q_\top)\}$,
- for any i and j, $\Delta(q_j, c_i) = \{(q_i, q_i)\}$ and $\Delta(q_j, d_i) = \{(q_i, q_\top), (q_\top, q_i)\}$.

Let $\mathcal{M}_n$ be the (nondeterministic) automaton (in Σ_n if n is even and in Π_n otherwise) whose set of states is $\{q_i \mid 2 \leq i \leq n\} \cup \{q'_i \mid 3 \leq i \leq n+1\} \cup \{q_\top\}$, where the rank of q_i and of q'_i is i and the rank of $q_\top$ is 2, and whose transition function Δ is defined as follows:

- for any i, $\Delta(q_\top, c_i) = \Delta(q_\top, d_i) = \{(q_\top, q_\top)\}$,
- for any i and any $s \neq q_\top$, $\Delta(s, c_i) = \{(s \star i, s \star i)\}$, and $\Delta(s, d_i) = \{(s \star i, q_\top), (q_\top, s \star i)\}$,

where $s \star i$ is defined as a function of s and i by the following table

	1	2	$\cdots$	i	$\cdots$	$n-1$	n
q_2	q'_3	q_2	$\cdots$	q_i	$\cdots$	q_{n-1}	q_n
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
q_j	q'_3	q_2	$\cdots$	q_i	$\cdots$	q_{n-1}	q_n
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
q_n	q'_3	q_2	$\cdots$	q_i	$\cdots$	q_{n-1}	q_n
q'_3	q'_3	q'_4	$\cdots$	q'_{i+2}	$\cdots$	q'_{n+1}	q_n
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
q'_j	q'_3	q'_4	$\cdots$	q'_{i+2}	$\cdots$	q'_{n+1}	q_n
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
q'_{n+1}	q'_3	q'_4	$\cdots$	q'_{i+2}	$\cdots$	q'_{n+1}	q_n

Let $W_n = L_{q_1}(\mathcal{W}_n)$ and $M_n = L_{q_2}(\mathcal{M}_n)$. One of them is in Π_n and the other one in Σ_n.

Proposition 4 *$W_n \cap K_n = M_n \cap K_n$.*

Proof A strategy σ in the games $G(\mathcal{W}_n, t)$ and $G(\mathcal{M}_n, t)$ consists in selecting one successor (left or right) at each node labelled by some d_i. Let t_σ be the (partial) tree obtained by cutting out the non-selected successors. With each branch b of t_σ we associate the infinite word $\bar{b} \in \{1, \ldots, n\}^\omega$ by substituting i for d_i or c_i. The strategy σ is winning if for each branch b of t_σ

$\mathcal{W}_n$: the largest number that occurs infinitely often in $\bar{b}$ is even,
$\mathcal{M}_n$: $\bar{b}$ is recognized by the parity word automaton whose transitions are given in the previous table (with q_2 as initial state).

It is easy to check that if $\bar{b}$ is in $\{1, \ldots, n\}^*(\{1, \ldots, n-1\}^\omega \cup \{2, \ldots, n\}^\omega)$ then these two conditions are equivalent. QED

An immediate consequence of this proposition is that $W_n \cup \overline{K_n} = M_n \cup \overline{K_n}$. Since $\overline{K_n} \in \Pi_2 \subseteq \Pi_n \cap \Sigma_n$, we get

Proposition 5 $W_n \cup \overline{K_n} = M_n \cup \overline{K_n} \in \Pi_n \cap \Sigma_n$ *and* $\overline{W_n} \cap K_n \in \Pi_n \cap \Sigma_n$.

3.2 The diagonal argument

Let us assume that $\overline{W_n} \cap K_n$ is in $\Gamma_{n-1} \subseteq \Pi_n \cap \Sigma_n$.

There is an automaton $\mathcal{A}$ such that $\overline{W_n} \cap K_n = L_{x_\bullet}(\mathcal{A})$, and for each $\succeq$-equivalence class X', $\rho(X')$ is included in $\{1, \ldots, n-1\}$ or in $\{2, \ldots, n\}$.

With each tree t over A_n and each state x of $\mathcal{A}$ we associate the tree $G_x(t)$ over A_n defined as follows. Let i be the rank of x, and let $t = f(t', t'')$ with $f \in A_n$. Let $\Delta(x, f) = \{(X'_1, X''_1), \ldots, (X'_k, X''_k)\}$. Then

$$\begin{aligned} G_x(t) = d_i(c_i(G^i_{X'_1}(t'), G^i_{X''_1}(t'')), \\ d_i(c_i(G^i_{X'_2}(t'), G^i_{X''_2}(t'')), \cdots \\ d_i(c_i(G^i_{X'_k}(t'), G^i_{X''_k}(t'')), c_i(G^i_{X'_k}(t'), G^i_{X''_k}(t''))) \cdots)) \end{aligned}$$

where, for all t, i, and $Y = \{x_1, \ldots, x_k\}$,

$$G^i_Y(t) = c_i(G_{x_1}(t), c_i(G_{x_2}(t), \cdots c_i(G_{x_k}(t), G_{x_k}(t)) \cdots)).$$

It is proved in [1] (or in [4]) that this mapping has the following property:

Proposition 6 $t \in L_x(\mathcal{A})$ *if and only if* $G_x(t) \in W_n$.
Each mapping G_x *has a unique fixed point* t_x.

Moreover, because $\mathcal{A}$ is in Γ_{n-1}, we have the additional property:

Proposition 7 *For any* t *and any* x, $G_x(t)$ *is in* K_n.

It follows that $t_{x_\bullet} \in \overline{W_n} \cap K_n$ if and only $t_{x_\bullet} \in W_n$. Since $t_{x_\bullet} \in K_n$, $t_{x_\bullet} \in \overline{W_n} \cap K_n$ if and only $t_{x_\bullet} \in \overline{W_n}$, a contradiction.

To extend this result to the modal μ calculus, let us consider the set of all directed graphs in which each vertex is labelled by a subset of A_n and each edge is labelled by ℓ or r. Let us consider the modal automaton $\mathcal{W}'_n$ having the same states as $\mathcal{W}_n$ and whose rules are

$$\Delta(q_j) = \bigwedge_{i=1}^{n} (c_i \Rightarrow (\langle \ell \rangle q_i \wedge \langle r \rangle q_i)) \wedge (d_i \Rightarrow ((\langle \ell \rangle q_i \wedge \langle r \rangle q_\top) \vee (\langle \ell \rangle q_\top \wedge \langle r \rangle q_i))).$$

It is easy to see that the set of binary trees accepted by this automaton is exactly W_n. More generally, with every tree automaton $\mathcal{A}$ one can associate a modal automaton $\mathcal{A}'$ in the same class, such that $L(\mathcal{A})$ is exactly the set of binary trees accepted by $\mathcal{A}'$. In a similar way, a modal automaton $\mathcal{K}'_n$ which accepts the language K_n is constructed. It is then proved that the modal automaton corresponding to the logical formula $\neg\mathcal{W}'_n \wedge \mathcal{K}'_n$ is equivalent both to a modal automaton in the class Π_n and to a modal automaton in Σ_n. Observe that the equivalence that we consider – and which is required for the argument – is now up to arbitrary transition systems, and not only up to the binary complete trees.

Now, if $\overline{W_n} \cap K_n$ is the set of trees accepted by a modal automaton $\mathcal{B}'$ in $comp(\Pi_{n-1}, \Sigma_{n-1})$ one can use the same diagonalisation technique: there exists G such that for any t, $G(t) \in K_n$ and $t \in \overline{W_n} \cap K_n$ if and only if $G(t) \in W_n$.

4 The separation theorem

We say that a language L is in $nd\Pi_n$ if there is a nondeterministic automaton $\mathcal{A}$ in Π_n and a state x such that $L = L_x(\mathcal{A})$.

Although $\Pi_n = nd\Pi_n$ for $n = 2$ [2, 4] this equality is no longer true for $n > 2$. We are going to show:

Theorem 8 *Let L and L' be two disjoint tree languages over an alphabet F. If both are in $nd\Pi_n$ (with $n \geq 2$) then there exists $K \in \Gamma_{n-1}$ such that $L \subseteq K \subseteq \overline{L'}$.*

We give the proof when the alphabet F has only binary symbols. The generalization to any alphabet is straightforward.

4.1 Run of an automaton

Given a nondeterministic automaton $\mathcal{A} = \langle X, \Delta, \rho\rangle$ and a tree t a run of $\mathcal{A}$ from state x on t is a mapping $\theta : \{l, r\}^* \to X$ such that $\theta(\varepsilon) = x$ and for any $u \in \{l, r\}^*$, $\langle\theta(ul), \theta(ur)\rangle \in \Delta(\theta(u), t(u))$.

We say that a run θ on t accepts t if for every $b = d_1 d_2 \cdots d_i \cdots \in \{l, r\}^\omega$, $\limsup_i \rho(\theta(d_1 \cdots d_i))$ is even.

A tree t belongs to $L_x(\mathcal{A})$ if and only if there is a run θ from x on t which accepts t.

4.2 A game for deciding nonemptiness

Let $\mathcal{A} = \langle X, \Delta, \rho\rangle$ and $\mathcal{A}' = \langle Y, \Delta', \rho'\rangle$ be two nondeterministic automata over an alphabet F of binary symbols.

Let us consider the (biparity) game $G(\mathcal{A}, \mathcal{A}')$ defined as follows.

- Eva's position are all the triples $(x, y, d) \in X \times Y \times \{l, r\}$.
- Adam's position are all the triples (r, r', f) where r is a rule of $\mathcal{A}$, r' a rule of $\mathcal{B}$, and $f \in F$.
- There is a move from (x, y, d) to (r, r', f) if and only if $r \in \Delta(x, f)$ and $r' \in \Delta'(y, f)$.
- If $r = (x', x'')$ and $r' = (y', y'')$ then from (r, r', f) there is a move to (x', y', l) and a move to (x'', y'', r).

Because $\Delta(x, f)$ and $\Delta'(y, f)$ are never empty, all maximal plays in this game are infinite. A play is winning for Eva if the sequence $(x_1, y_1, d_1), (x_2, y_2, d_2), \ldots,$ $(x_i, y_i, d_i), \ldots$ of Eva's positions along this play is such that both $\limsup \rho(x_i)$ and $\limsup \rho'(y_i)$ are even.

The proof of the following result is quite easy and can be found in [6].

Proposition 9 *$L_x(\mathcal{A}) \cap L_y(\mathcal{A}')$ is not empty if and only if for some d the position (x, y, d) is winning for Eva.*

Note that the sets of plays from (x, y, l) and (x, y, r) are the same (except, of course, for the first position). Therefore one of these positions is winning if and only if the other is.

By the previous proposition, for any two states x and y, $L_x(\mathcal{A}) \cap L_y(\mathcal{A}') = \emptyset$ if and only if the position (x, y, l) in $G(\mathcal{A}, \mathcal{A}')$ is not winning for Eva. In this case Adam has a winning (winning for him!) strategy with finite-memory H, i.e., in any position (r, r', f) he chooses either the left or the right direction.

This strategy yields a finite graph G whose nodes have the form (x, y, d, h) or (r, r', f, h). If (x, y, d, h) is a node of G then for any $f \in F$, any $r \in \Delta(x, f)$, any $r' \in \Delta'(y, f)$, there is an h' such that (r, r', f, h') is a successor of (x, y, d, h) in G. If (r, r', f, h') is a node of G and if $r = (x_l, x_r)$, $r' = (y_l, y_r)$ then this node has a unique successor (x_d, y_d, d, h'') for some $d \in \{l, r\}$ and some $h'' \in H$. We denote by $succ(s, r, r', f)$ the pair $\langle s', d' \rangle$ where s' is the unique successor of the successor (r, r', f, h') of s and where d' is the direction of s'.

Let S be the set of nodes of G of the form (x, y, d, h). For any $s = (x, y, d, h) \in S$ we set $\pi_X(s) = x$, $\pi_Y(s) = y$, and $\pi_D(s) = d$.

Proposition 10 *For any infinite path p in G, the projection $s_1, s_2, \ldots, s_i, \ldots$ of p on S is such that $\limsup_i \rho(\pi_X(s_i))$ is odd or $\limsup_i \rho'(\pi_Y(s_i))$ is odd.*

Moreover, let $s = (x, y, d, h)$ be in S, let t be any tree, θ be any run of $\mathcal{A}$ from x on t and θ' be any run of $\mathcal{A}'$ from y on t. These four data define a unique path in G. We denote by $b(t, s, \theta, \theta')$ the projection $s = s_1, s_2, \ldots, s_i, \ldots$ on S of this unique path. Then for any i, $\pi_X(s_i) = \theta(\pi_D(s_1) \cdots \pi_D(s_{i-1}))$ and $\pi_Y(s_i) = \theta'(\pi_D(s_1) \cdots \pi_D(s_{i-1}))$.

It follows that for any x and y, there exist d and h such that $s = (x, y, d, h)$ is in S, if and only if $L_x(\mathcal{A}) \cap L_y(\mathcal{A}') = \emptyset$.

4.3 The separation property

Let $\mathcal{A}$ and $\mathcal{A}'$ be two nondeterministic automata in Π_{n+1}. Without loss of generality, we may assume that there is an even m such that $\rho(X)$ and $\rho'(Y)$ are both included in $\{m - n, \ldots, m\}$.

Let us consider the subgraph G of $G(\mathcal{A}, \mathcal{A}')$ induced by a winning strategy of Adam, defined in the previous section Let us define the preorder $\succeq$ on S by $s \succeq s'$ if and only if there is a path from s to s'. It is easy to see that s is equivalent to s' (with espect to the equivalence induced by the preorder $\succeq$) if and only if they belongs to the same strongly connected component of G.

We define a new mapping $\rho'' : S \to \mathbb{N}$ as follows. Let C be a strongly connected component in G, which contains at least one node of S. If C is trivial (it contains only one s) then we set $\rho''(s) = m - 1$. If C is nontrivial there cannot be in C an s and an s' with $\rho(\pi_X(s)) = \rho'(\pi_Y(s')) = m$. Therefore either ρ never has the value m on $\pi_X(C)$ or ρ' never has the value m on $\pi_Y(C)$. In the first case we set $\rho''(s) = \rho(\pi_X(s))$. In the second case we set $\rho''(s) = \rho'(\pi_Y(s)) + 1$.

Proposition 11 *Let $s_1, s_2, \ldots, s_i, \ldots$ be the projection on S of an infinite path in G. Then*
$\limsup_i \rho(\pi_X(s_i))$ *is even* $\Rightarrow \limsup_i \rho''(s_i)$ *is even* $\Rightarrow \limsup_i \rho'(\pi_Y(s_i))$ *is odd* .

Proof Let $k = \limsup_i \rho(\pi_X(s_i))$, $k' = \limsup_i \rho'(\pi_Y(s_i))$, and $k'' = \limsup_i \rho''(s_i)$

Since from some n, the set $\{s_i \mid i \geq n\}$ is included in a nontrivial strongly connected component of G, either $k'' = k$, or $k'' = k' + 1$. If k is even, then by

Proposition 10, k' is odd, thus k'' is always even. If k' is even, then k is odd, thus k'' is always odd.

QED

We define two alternating automata $\mathcal{C}_1 = \langle S, \Delta_1'', \rho'' \rangle$ and $\mathcal{C}_2 = \langle S, \Delta_2'', \rho'' \rangle$ by

- $\Delta_1''(s, f) = \bigvee_{r \in \Delta(\pi_X(s), f)} \bigwedge_{r' \in \Delta'(\pi_Y(s), f)} succ(s, r, r', f)$.
- $\Delta_2''(s, f) = \bigwedge_{r' \in \Delta'(\pi_Y(s), f)} \bigvee_{r \in \Delta(\pi_X(s), f)} succ(s, r, r', f)$.

Proposition 12 *$\mathcal{C}_1$ and $\mathcal{C}_2$ are in $comp(\Pi_n, \Sigma_n)$.*
For any s, $L_s(\mathcal{C}_1) \subseteq L_s(\mathcal{C}_2)$.

Proof Using the preorder $\succeq$ on S, and the definition of ρ'', it is easy to see that $\mathcal{C}_1$ and $\mathcal{C}_2$ are in $comp(\Pi_n, \Sigma_n)$.

Since the boolean formula

$$\bigvee_{r \in \Delta(\pi_X(s), f)} \bigwedge_{r' \in \Delta'(\pi_Y(s), f)} succ(s, r, r', f)$$

logically implies

$$\bigwedge_{r' \in \Delta'(\pi_Y(s), f)} \bigvee_{r \in \Delta(\pi_X(s), f)} succ(s, r, r', f),$$

we obviously have $L_s(\mathcal{C}_1) \subseteq L_s(\mathcal{C}_2)$. QED

Note that if we exchange the roles of $\mathcal{A}$ and $\mathcal{A}'$ in the above construction we get two automata $\mathcal{C}_1'$ and $\mathcal{C}_2'$. The dual automaton $\widetilde{\mathcal{C}_2}$ of $\mathcal{C}_2$, which satisfies $L_s(\widetilde{\mathcal{C}_2}) = \overline{L_s(\mathcal{C}_2)}$, is the automaton $\langle S, \widetilde{\Delta_2''}, \rho_{+1}'' \rangle$ where $\rho_{+1}''(s) = \rho''(s) + 1$ and

$$\widetilde{\Delta_2''}(s, f) = \bigvee_{r' \in \Delta'(\pi_Y(s), f)} \bigwedge_{r \in \Delta(\pi_X(s), f)} succ(s, r, r', f).$$

It is easy to see that $\widetilde{\mathcal{C}_2}$ is equivalent to $\mathcal{C}_1'$. Similarly, $\widetilde{\mathcal{C}_1}$ is equivalent to $\mathcal{C}_2'$.

Therefore, the following proposition achieves the proof of the Separation Theorem.

Proposition 13 *For any $s \in S$, $L_{\pi_X(s)}(\mathcal{A}) \subseteq L_s(\mathcal{C}_1)$.*

Proof If $t \in L_{\pi_X(s)}(\mathcal{A})$ there exists an accepting run θ from $\pi_X(s)$ on t. For any run θ' from $\pi_Y(s)$ let $b(t, s, \theta, \theta') = s_1, \ldots, s_i, \ldots$. By Proposition 10, $\limsup_i \rho(\pi_X(s_i))$ is even, hence, by Proposition 11, $\limsup_i \rho''(s_i)$ is even. In the game $G(\mathcal{C}_1, t)$, an Eva's strategy consists in selecting $r \in \Delta(\pi_X(s), t(u))$ at each node (s, u). If Eva chooses the rule r that θ uses at node u, then, by the previous remark, this strategy is winning at (s, ε). Hence, $t \in L_s(\mathcal{C}_1)$.

QED

4.4 The case of modal μ-calculus

If instead of tree automata we consider modal automata, we get the same result using a similar proof.

The modal automata we are considering are intended to accepts Kripke structures over a set of local properties. Edges are not labelled: The case of labelled edges can be treated in exactly the same way. One can assume without loss of generaliy that each node u of a Kripke structure K has a unique label $\lambda(u)$ taken from the powerset F of local properties. Finally, instead of using the usual modalities $\langle\rangle$ and $[]$, we use the unique modality $\rightarrow$, introduced in [7], whose argument is a set, possibly empty, of variables (for nondeterministic automata) or a set, possibly empty, of boolean combinations of variables (for alternating automata) with the following interpretation: a vertex v of a labelled graph satisfies $\rightarrow E$ if for any successor v' of v there is an $e \in E$ such that v' satisfies e, and for any $e \in E$ there is a successor v' of v such that v' satisfies e. It should be noted that the conjunction $\rightarrow E \wedge \rightarrow E'$ is equivalent to the disjunction of the terms $\rightarrow \{e \wedge e' \mid (e, e') \in R\}$ for all $R \subseteq E \times E'$ such that the first and second projections of R are respectively E and E'.

Therefore, a nondeterministic modal automaton associates with each state $x \in X$ and each symbol $f \in F$ a set $\Delta(x, f)$ of rules, possibly empty, where each rule r is $\rightarrow X_r$ where X_r is a subset of X.

The game $G(\mathcal{A}, \mathcal{A}')$ for deciding emptyness is defined as follows:

- In position (x, y) Eva chooses a symbol f, a rule r in $\Delta(x, f)$ and a rule r' in $\Delta'(y, f)$ and moves to (r, r', f).
- In position (r, r', f) Eva chooses a relation $R \subseteq X_r \times Y_{r'}$ such that its projections are X_r and $Y_{r'}$ and moves to R. Note that if $X_r = Y_{r'} = \emptyset$ there exists only one such relation, the empty one. If only one of these two sets is empty, there is no such relation, therefore the position (r, r', f) is loosing for Eva.
- In position R, Adam moves to $(x', y') \in R$. If R is the empty relation, this position is loosing for Adam.

Let G be the subgraph of $G(\mathcal{A}, \mathcal{A}')$ induced by a winning strategy of Adam. For any s, any f, any $r \in \Delta(\pi_X(s), f)$, any $r' \in \Delta'(\pi_Y(s), f)$, there exists in G a unique successor (r, r', f, h') of s. Note that G does not contain any node (r, r', f, h) with $X_r = Y_{r'} = \emptyset$ because this node has a unique successor which is a loosing Adam's position. Let $succ(s, r, r', f)$ be the set of successors of (r, r', f, h'). This set is empty if one of the two sets X_r or $Y_{r'}$ is empty. Otherwise, this set has the following property:

Proposition 14 *Let $R \subseteq X_r \times Y_{r'}$ be $\{(\pi_X(s'), \pi_Y(s')) \mid s' \in succ(s, r, r', f)\}$. There exists x in X_r such that for any y in $Y_{r'}$, $(x, y) \in R$ or there exists y in $Y_{r'}$ such that for any x in X_r, $(x, y) \in R$.*

Proof Assume that it is not true. Then the projections of the relation $(X_r \times Y_{r'}) - R$ are X_r and $Y_{r'}$. Therefore $succ(s, r, r', f)$ must contain s' such that $(\pi_X(s'), \pi_Y(s')) \notin R$, a contradiction. QED

Let us add to the set S two fresh elements $\top$ and $\bot$. For each node s of G and for each node (r, r', f, h) which is a successor of s, we define the modal expression $M(s, r, r', f)$ as follows.

- If X_r and $Y_{r'}$ are not empty and if $succ(s, r, r', f)$ satisfies the first condition of Proposition 14 (for some x) then $M(s, r, r', f) = \langle\rangle \bigwedge\{s' \in succ(s, r, r', f) \mid \pi_X(s') = x\}$.
- If X_r and $Y_{r'}$ are not empty and if $succ(s, r, r', f)$ satisfies the second condition (for some y) then $M(s, r, r', f)$ is equal to the (dual) modal expression $[] \bigvee\{s' \in succ(s, r, r', f) \mid \pi_Y(s') = y\}$.
- If $X_r = \emptyset$ then $M(s, r, r', f) = []\bot$.
- If $Y_{r'} = \emptyset$ then $M(s, r, r', f) = \langle\rangle\top$.

As in the case for trees, we define $\rho'' : S \to \mathbb{N}$. We define the alternating automata $\mathcal{C}_1 = (S', \Delta_1'', \rho'')$ and $\mathcal{C}_2 = (S', \Delta_2'', \rho'')$ in $comp(\Pi_n, \Sigma_n)$, where $S' = S \cup \{\bot, \top\}$, as follows.

- $\bot$ is a state which accepts nothing: its rank $\rho''(\bot)$ is any value and for any f, $\Delta_1''(\bot, f) = \Delta_1''(\bot, f) = \emptyset$.
- $\top$ is a state which accepts everything: its rank $\rho''(\top)$ is the even number $m-2$ and for any f, $\Delta_1''(\top, f) = \Delta_1''(\top, f) = []\top$.
- $\Delta_1''(s, f) = \bigvee_{r \in \Delta(\pi_X(s), f)} \bigwedge_{r' \in \Delta'(\pi_Y(s), f)} M(s, r, r', f)$,
- $\Delta_2''(s, f) = \bigwedge_{r' \in \Delta'(\pi_Y(s), f)} \bigvee_{r \in \Delta(\pi_X(s), f)} M(s, r, r', f)$.

Note that in the above definition, a union over an empty set is equal to $\bot$ and an intersection over an emptyset is equal to $\top$.

Here again, the dual of $\mathcal{C}_2$ is the automaton $\mathcal{C}_1'$ obtained as $\mathcal{C}_1$ but exchanging the role of $\mathcal{A}$ and $\mathcal{A}'$. Hence, for proving the Separation Theorem, it is enough to prove Proposition 13. Indeed Proposition 11 still holds. As to Proposition 10, we have to substitute the notion of (winning) strategy of a nondeterministic modal automaton $\mathcal{A}$ (or $\mathcal{A}'$) on a Kripke structure K for the notion of (accepting) run on a tree.

Let K be a Kripke structure. For any node u of K, we denote by $u*$ the set of successors of u, which is possibly empty. A (positional) strategy σ is a mapping which associate with every pair (x, u) a rule $r \in \Delta(x, \lambda(u))$ (this implies that $\Delta(x, \lambda(u))$ is not empty) and with every pair (X_r, u) a relation in the set $X_r \leftrightarrow u*$ of all relations included in $X_r \times u*$ whose first and second projection are X_r and $u*$ (this implies that X_r is empty if and only if $u*$ is empty).

For any (x, u), this strategy defines a set $P_\sigma(x, u)$ of paths $(x, u) = (x_1, u_1)(x_2, u_2)(x_3, u_3)\ldots$ such that its projection $\pi_K(P_\sigma(x, u))$ on K is the set of all maximal paths in K starting in u. The strategy is winning if all infinite sequences in $\pi_X(P_\sigma(x, u))$ satisfy the parity condition with respect to ρ.

Let us assume that σ is a strategy for $\mathcal{A}$ winning at $(\pi_X(s), u)$. We construct a strategy for Eva in the game $G(\mathcal{C}_1, K)$ winning at (s, u).

Let us assume that Adam plays according to a given arbitrary strategy in the game $G(\mathcal{C}_1, K)$ and the play consistent with the two strategies has reached a position (s_i, u_i).

Eva selects r_i in the nonempty set $\Delta(\pi_X(s_i), \lambda(u_i))$ and a relation $R_i \in X_{r_i} \leftrightarrow u_i*$. If $\Delta'(\pi_Y(s_i), \lambda(u_i)) = \emptyset$ then $\Delta_1''(s_i, \lambda(u_i)) = \top$ and Eva wins. Otherwise Adam chooses $r_i' \in \Delta'(\pi_Y(s_i), \lambda(u_i))$. Next:

- If X_{r_i} is empty (and also u_i*) then $M(s_i, r_i, r_i', \lambda(u_i)) = []\bot$ and Eva wins, since u_i* is empty.
- If X_{r_i} is not empty and $Y_{r'i}$ is empty, then $M(s_i, r_i, r_i', \lambda(u_i)) = \langle\rangle\top$ and Eva wins, since u_i* is not empty.
- If $M(s_i, r_i, r_i', \lambda(u_i)) = \langle\rangle \bigwedge\{s' \in succ(s_i, r_i, r_i', \lambda(u_i)) \mid \pi_X(s') = x\}$, Eva chooses $u_{i+1} \in u_i*$ such that $(x, u_{i+1}) \in R_i$ and Adam chooses s_{i+1} (note that $\pi_X(s_{i+1}) = x$).
- If $M(s, r, r', f) = [] \bigvee\{s' \in succ(s_i, r_i, r_i', \lambda(u_i)) \mid \pi_Y(s') = y\}$, Adam chooses $u_{i+1} \in u_i*$ and Eva chooses s_{i+1} such that $(\pi_X(s_{i+1}), u_{i+1}) \in R_i$.

By construction the play $(s, u) = (s_1, u_1), (s_2, u_2), \ldots$ satisfies

$$(\pi_X(s_1), u_1), (\pi_X(s_2), u_2), \ldots \in P_\sigma(\pi_X(s), u).$$

Thus, by Proposition 11, it is won by Eva.

References

1. A. Arnold. The μ-calculus alternation-depth hierarchy is strict on binary trees. *RAIRO-Theoretical Informatics and Applications*, 33:329–339, 1999.
2. A. Arnold and D. Niwiński. Fixed point characterization of Büchi automata on infinite trees. *J. Inf. Process. Cybern. EIK*, 26:453–461, 1990.
3. A. Arnold and D. Niwiński. Fixed point characterization of weak monadic logic definable sets of trees. In M. Nivat and A. Podelski, editors, *Tree automata and Languages*, pages 159–188. Elsevier, 1992.
4. A. Arnold and D. Niwiński. *Rudiments of μ-calculus.* Number 146 in Studies in logic and the foundations of mathematics. Elsevier, North-Holland, 2002.
5. A. Arnold and L. Santocanale. Ambiguous classes in the games μ-calculus hierarchy. In *FOSSACS 2003 (to appear).*
6. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, to appear, 2002.
7. D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. *Lecture Notes in Computer Science*, 969:552–562, 1995.
8. R. Kaivola. On modal mu-calculus and Büchi tree automata. *Information Processing Letters*, 54:17–22, 1995.
9. O. Kupferman and M. Y. Vardi. The weakness of self-complementation. *Lecture Notes in Computer Science*, 1563:455–466, 1999.
10. D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of trees and its complexity. *Theoretical Computer Science*, 97:233–244, 1992.
11. M. O. Rabin. Weakly definable relations and special automata. In Y. Bar-Hillel, editor, *Mathematical Logic and Foundation of Set Theory*, pages 1–23. North–Holland, Amsterdam, 1990.

A Fixpoint Logic for Labeled Markov Processes

Vincent Danos[*] CNRS & Université Paris 7

Josée Desharnais Université Laval, Québec

March 18, 2003

We develop in this abstract a probabilistic fixpoint logic for Labeled Markov Processes (LMPs). One reason for doing this comes from [2, 3]. There, it was shown that the LMP logic characterizing bisimulation can be used to define in a natural way finite-state approximants of LMPs. An extension of this logic with fixpoints such as the one we propose allows for stronger notions of approximants. Steady properties, *i.e.* properties related to infinite behaviours, can be obtained in finite approximations.

Our logic only deals with greatest fixpoints. In a probabilistic setting, one has a the pending constraint that all logical terms denote measurable sets. Since measurability is preserved by *countable* boolean operations, only continuous or cocontinuous operators are meaningful. Least fixpoints are trivial in our case (more about this below) so we're left with greatest fixpoints.

As an illustration of the descriptive power of the logic, we provide an explicit construction of the coarsest probabilistic simulation of a given finite LMP by an arbitrary one. This construction is interesting in its own right. Finally a continuous state space example is given.

An LMP can be described as a family of probabilities $(p(s))_{s\in S}$ indexed by the state space S, $p(s)(A)$ representing the probability that the process will jump from s to A a measurable subset of S. In some special circumstances (when all the $p(s)$ are mutually absolutely continuous, *i.e.* define the same negligible events) the Radon-Nikodým theorem makes it possible to extend the σ-algebra of events into a complete boolean algebra and therefore a logic with both fixpoints and arbitrary monotone operators seems possible. We might pursue this option in the future.

[*] *Corresponding author*: Équipe PPS, Université Paris 7 Denis Diderot, Case 7014, 2 Place Jussieu 75251 PARIS Cedex 05, Vincent.Danos@pps.jussieu.fr

1 Preliminaries

Definition 1 (LMP) $\mathcal{S} = (S, \Sigma, h : L \times S \times \Sigma \to [0,1])$ *is a Labelled Markov Process (LMP) if* (S, Σ) *is a measurable space, for all* $a \in L$, $A \in \Sigma$, $h(a,s,A)$ *is* Σ*-measurable as a function of* s *and for all* $s \in S$, $h(a,s,A)$ *is a subprobability as a function of* A.

Some particular cases: 1) when S is finite and $\Sigma = 2^S$ we have the familiar probabilistic automaton, 2) when $h(a,s,A)$ doesn't depend on s or on a we have the familiar (sub)probability triple. An example of the latter situation is $([0,1], \mathcal{B}, h)$ with $h(a,s)(B) = \lambda(B)$ with λ the Lebesgue measure on Borelians.

Definition 2 (shifts) *For* $a \in L$, $r \in [0,1]$, *one defines endomaps of* Σ, shifts *and* strict shifts *as:*

$$\langle a \rangle_r(A) = \{s \mid h(a,s)(A) \geq r\} \qquad \{a\}_r(A) = \{s \mid h(a,s)(A) > r\}$$

Shifts are cocontinuous and strict shifts are continuous, but they have the empty set as least fixpoint: $\{a\}_r(\varnothing) = \varnothing$. Strict shifts are not co-continuous, neither are shifts continuous.

As an example, consider again $([0,1], \mathcal{B}, h)$ as above:

$$\begin{array}{lcl} \{a\}_0(\downarrow (0,1/n] = \varnothing) = \varnothing & \subset & \downarrow \{a\}_0((0,1/n]) = [0,1] \\ \uparrow \langle a \rangle_1((0,1-1/n]) = \varnothing & \subset & \langle a \rangle_1(\uparrow (0,1-1/n] = (0,1]) = [0,1] \end{array}$$

2 Fixpoint logic

Syntax. Let a countable set of variables x, y, etc. be given. Fixpoint terms are given by the following grammar:

$$t := x \mid \top \mid t \cap t \mid t \cup t \mid \langle a \rangle_r t \mid \langle t, \ldots, t \rangle \mid \pi_i t \mid \nu x.t$$

Examples of (closed) terms: $\nu x.\langle a \rangle_{.5} x$, $\nu x.\langle \langle a \rangle_{.5} \pi_2 x, \langle a \rangle_{.2} \pi_1 x \rangle$. There is an evident typing discipline and we will assume that all terms are well-typed over a base type σ. Types will be written as $[\sigma^n, \sigma^m]$. Also, terms will be considered up to the usual product equations.

Interpretation. Let **Ccpo** stand for the Cartesian category of ω-cocontinuous functions between ω-cocomplete pointed partial orders.

For any LMP $\mathcal{S} = (S, \Sigma, h)$ we denote by $\mathbf{C}_\mathcal{S}$ the full subcategory of **Ccpo** generated by Σ. Given a term t of type $[\sigma^n, \sigma^m]$ one defines inductively $[\![t]\!]_\mathcal{S} \in \mathbf{C}_\mathcal{S}[\Sigma^n, \Sigma^m]$. All clauses are trivial, except:

$$[\![\nu y^m.t]\!]_\mathcal{S}(x^n) = \downarrow_p (\lambda y^m.[\![t]\!]_\mathcal{S}(x, y))^p(S, \dots, S).$$

The notation $\downarrow_p X_p$ means that the sequence X_p is decreasing in Σ^m (equipped with the product ordering). We observe that the fixpoint used here is the ambient greatest fixpoint of **Ccpo** which is a simple example of an iteration operator (as defined in [1]) and is uniform with respect to costrict maps (as defined in [4]).

Finite LMPs as terms. Given a subprobabilistic automaton $\mathcal{A} = (I, 2^I, k)$ with states $I = \{1, \dots, n\}$, we define $t_1(\mathcal{A})$ and $t(\mathcal{A})$ both of type $[\sigma^n, \sigma^n]$:

$$\pi_i(t_1(\mathcal{A})) = \bigcap_{j \in I, a \in L} \langle a \rangle_{k(a,i,\{j\})}(x_j)$$
$$\pi_i(t(\mathcal{A})) = \bigcap_{J \subseteq I, a \in L} \langle a \rangle_{k(a,i,J)}(\bigcup_{j \in J} x_j)$$

By construction, for any $\mathcal{S}$: $[\![t(\mathcal{A})]\!]_\mathcal{S} \leq [\![t_1(\mathcal{A})]\!]_\mathcal{S}$ (for pointwise ordering), because there are more terms in the intersection defining $\pi_i(t(\mathcal{A}))$; and therefore their respective fixpoints will be in the same order. Intermediate approximations where J is varying in a subset of Σ make perfect sense.

3 Simulations

If $\mathfrak{R}$ is a binary relation over S, and $s \in S$, we write $\mathfrak{R}(s)$ for $\{t \mid (s, t) \in \mathfrak{R}\}$. Likewise if $A \subset S$, $\mathfrak{R}(A) = \cup_{s \in A} \mathfrak{R}(s)$.

Definition 3 (probabilistic simulation) *Let two LMPs $\mathcal{S}_1$ and $\mathcal{S}_2$ be given, one says a relation $\mathfrak{R} \subseteq S_1 \times S_2$ is a simulation of $\mathcal{S}_1$ by $\mathcal{S}_2$ when:*

$$\forall (s_1, s_2) \in \mathfrak{R}, a \in L, A_1 \in \Sigma_1 :$$
$$\mathfrak{R}(A_1) \in \Sigma_2 \Rightarrow h_1(a, s_1, A_1) \leq h_2(a, s_2, \mathfrak{R}(A_1)).$$

The empty relation is a simulation, so the mere existence of a simulation is not conclusive. What matters is if a given state $s_1 \in S_1$ is *simulated* by a state $s_2 \in S_2$, that is to say, if there is a simulation $\mathfrak{R}$ with $(s_1, s_2) \in \mathfrak{R}$.

Proposition 4 *For any LMP $\mathcal{S}$ and finite LMP $\mathcal{A}$, there is a coarsest probabilistic simulation $\mathfrak{S}(\mathcal{A}, \mathcal{S}) \subseteq I \times S$ of $\mathcal{A}$ by $\mathcal{S}$ and it is given by:*

$$(i, s) \in \mathfrak{S}(\mathcal{A}, \mathcal{S}) := s \in \pi_i [\![\nu x.t(\mathcal{A})]\!]_\mathcal{S}.$$

The corresponding relation was defined with t_1 in [2] and shown to yield the coarsest non-deterministic simulation. Of course both relations will coincide when for each a, $\mathcal{A}$ has at most one non-zero a transition from each state. The difference only shows when there are correlations between transitions from a same state as in the following example.

A simple example. Suppose $L = \{a\}$ and let $\mathcal{S} = q_0 \rightarrow q_1$ and $\mathcal{A} = q'_0 \rightarrow q'_1, q'_2$ with all transitions having probability .5. In the *non-deterministic* interpretation given by t_1 of type $[\sigma^3, \sigma^3]$ we have:

$$[\![\pi_0 t_1(\mathcal{A})]\!]_S(S,S,S) = \langle a\rangle_{.5}(S) \cap \langle a\rangle_{.5}(S) = \{q_0\} \cap \{q_0\} = \{q_0\}$$
$$[\![\nu x.t_1(\mathcal{A})]\!]_S = [\![t_1(\mathcal{A})]\!]_S(S,S,S) = (\{q_0\}, S, S).$$

Now with the refined *probabilistic* interpretation of t:

$$[\![\pi_0 t(\mathcal{A})]\!]_S(S,S,S) = \langle a\rangle_{.5}(S) \cap \langle a\rangle_{.5}(S) \cap \langle a\rangle_{1.0}(S \cup S) = \varnothing,$$
$$[\![\nu x.t(\mathcal{A})]\!]_S = (\varnothing, S, S).$$

Hence, by the proposition above, no state of $\mathcal{S}$ simulates q'_0 (in the sense of definition 3). This can be seen directly: q'_0 in $\mathcal{A}$ has probability 1 to do something, while no state in S has probability more than .5 of doing something.

A continuous state example. Consider the continuous LMP, $L = \{a, b\}$, $h_a(s, A) = s\lambda(A)$ and $h_b(s, A) = (1-s)\lambda(A)$. Small s tend to be insensitive to as and hypersensitive to bs and conversely. We can compute the shifts on segments: $\langle a\rangle_\alpha([x,y]) = [f(y-x), 1]$, $\langle b\rangle_\beta([x,y]) = [0, g(y-x)]$ with $f(\xi) = \alpha/\xi$, and $g(\xi) = 1 - \beta/\xi$, and we see the set of (rational) segments is stable by (rational) shifts. Combining a and b in the temporal property gives:

$$[\![\langle a\rangle_\alpha\langle b\rangle_\beta \ldots \top]\!] = [\alpha_n, 1] \qquad [\![\langle b\rangle_\beta\langle a\rangle_\alpha \ldots \top]\!] = [0, 1-\beta_n]$$
$$\beta_{n+1} = \frac{\beta}{1-\alpha_n} \qquad \alpha_{n+1} = \frac{\alpha}{1-\beta_n}.$$

The α_n sequence converges to a root of $x^2 + \alpha - x(1 + \alpha - \beta)$. If we set $\alpha = \beta = 1/4 - \epsilon$, with $\epsilon > 0$, α_n converges to $1/2 \pm \sqrt{\epsilon}$ when ϵ is small enough to keep the sequences α_n and β_n within $[0,1]$. So we get two fixpoints, and the smallest gives the biggest segment:

$$[\![\nu x.(\langle a\rangle_\alpha(\pi_2 x), \langle b\rangle_\beta(\pi_1 x))]\!] = ([0, 1/2 + \sqrt{\epsilon}], [1/2 - \sqrt{\epsilon}, 1])$$

Observe the symmetry in the solution. Our fixpoint does find a pair of state sets in the right hand side implementing the two-state subprobabilistic automaton (associated to the term) given in the lefthand side.

References

[1] S. Bloom and Z. Ésik. Iteration theories. *EATCS Monographs on Theoretical Computer Science*, 1993.

[2] Vincent Danos and Josée Desharnais. Labeled Markov processes: stronger and faster approximations. Submitted, 2003.

[3] J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Approximating continuous Markov processes. *Information and Computation*, 2003. To appear. Available from `http://www.ift.ulaval.ca/jodesharnais`.

[4] Alex Simpson and Gordon Plotkin. Complete axioms for categorical fixed-point operators. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS 2000)*, pages 30–41. IEEE, June 2000.

A Bisimilarity Logical Relation for the Object Calculus S

Luis Dominguez[*]
Departamento de Matematica, Instituto Superior Tecnico
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
lald@math.ist.utl.pt

Abstract

In this paper I define a bisimilarity logical relation and prove it equal to the axiomatic operational congruence of the primitive object calculus S [1] of Abadi and Cardelli, observing termination at every type. The paper also summarizes essential theory of the Galois operators underlying the approach of Andrew Pitts adapted here. As in [3], I define an open type indexed family of relators over closed term relations, in terms of auxiliary relators returning closed value relations, and extend them substitutively to logical relations between open terms.

1 Introduction

I have been developing a (co)inductive relational theory for reasoning, specifying and verifying typed applicative sequential object based programs "desugared" as terms of a calculus.

Among the many such foundational calculi in the literature, modelling applicative and imperative object oriented programming languages, I preferred those studied by Abadi and Cardelli in their book [1] for their better understood type systems and meta theory. Abadi and Cardelli studied a number of object calculi, with increasingly expressive and complex type systems, namely: "first-order" (at most recursive types), "second-order" (with type quantifiers) and "higher-order" calculi (with type operators). Due to the shortcomings of the first-order calculi [1] and the syntactical and meta-theoretic complexity of the higher-order calculi and subtyping systems (for example involving type conversion), I eventually concentrated on second-order primitive object calculi. From these, quite expressive albeit with heavier syntax and (sub)typing rules than standard typed lambda calculi, I chose the largest S_λ [2]. This may be seen as a combination of two subcalculi: $S_\forall$ and S_{cbv}. $S_\forall$ extends the smallest subcalculus S with subtype

[*]Thanks to António Ravara for supervision, feedback and discussions on this subject.

bounded polymorphism. I call S_{cbv} the extension of S with call by value functions. Abadi and Cardelli introduced S as the essence of applicative typed object-based programming, and its extension $S_\forall$ also for class-based and polymorphic programming. The inclusions between such subcalculi of S_λ may be drawn as follows.

$$\begin{array}{ccc} S_\forall & \subset & S_\lambda \\ \cup & & \cup \\ S & \subset & S_{cbv} \end{array}$$

Gordon defined an experimental similarity relation and proved by Howe's method it to be preadequate (observing termination at every type), subsumptive, compatible, substitutive and transitive. He added call by value functions (obtaining S_{cbv} or S_λ), so that experimental similarity equals not only the calculus operational precongruence but also its contextual precongruence; instead of being finer grained than the latter.

Inspired by Gordon's coincidence between the contextual precongruence and the experimental similarity for S_λ, I first solved the problem of finding a logical relation provably equal to the operational congruence for S_λ. Such type indexed family of relations constitutes a relational parametric model of the calculus and gives further insight on the meta theory of S_λ. This extensionality result expands the calculus theory in a richer relational framework, allowing practical reasoning about term and stack relations to prove behavioural properties, as illustrated by Pitts [4].

To prove the extensionality theorem, I adopt the approach of Andrew Pitts, first explicit in [4] and [5] for call by name calculi, and tailored in [3] for a call by value counterpart of PCF with let, record, polymorphic and existential types. As in [3], I define an open type indexed family of relators over closed term relations, in terms of auxiliary relators returning closed value relations, and extend them substitutively to logical relations between open terms. Subtyping and recursive object types are novel aspects relatively to the referred work of Pitts, for which I had to adapt his method and needed to introduce meta notions and notation, such as "bond".

In order to reveal the method and essence of the extensionality proof in a shorter and more clear way, this paper is confined to the simplest subcalculus S of S_λ, which happens to keep most interesting proof cases. Despite its few primitives it is Turing complete, as PCF could be encoded in it based on function and fixpoint encodings in [1] which also shows how to regard it as a kernel applicative typed object-based programming language. But unlike for S_λ, it is not known if its bisimilarity is equal or contained in its contextual congruence. If value and type substitution is representable by contexts of S the encodings are probably too intrincate to be useful.

The contributions in this paper are:

(1) adaptation of Pitts' approach, namely meta notions and notation, to

cope with the novel aspects of subtyping and primitive covariant self (recursive) object types;
(2) a synopsis of essential theory of the Galois operators underlying Pitts' approach [4];
(3) definitions of value bisimilarity, and (term) bisimilarity, logical relations which use maximum fixed points of non trivial typed relators, namely for each object type constructor;
(4) proof that bisimilarity is equal to the operational congruence of S.

2 Notation

We use an isomorphic variant of S whose syntax is given by the following grammar.

$$\begin{array}{rcl}
\text{variance } \vartheta & ::= & \circ \mid + \mid - \\
\text{type } t & ::= & u \mid \mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle \\
\text{term } e & ::= & v \mid e.l \mid e\,r \\
\text{value } v & ::= & z \mid o \\
\text{object } o & ::= & \mathsf{ob}(u = t)\langle_{j\in L} j = \varsigma m_j\rangle \\
\text{redefine frame } r & ::= & \langle l{:=}(u{<:}t, z : u)\varsigma m\rangle \\
\text{method } m & ::= & (z : u)e
\end{array}$$

top abbreviates $\mathtt{Ob}(u)\langle\rangle$ in this calculus with only object types.
The next grammar defines the syntax of some meta notation.

$$\begin{array}{rcl}
\text{frame } f & ::= & .l \mid r \\
\text{frame stack } s & ::= & \varepsilon \mid fs \\
\text{base } \Gamma & ::= & \varepsilon \mid \Gamma, u{<:}t \mid \Gamma, z : t \\
\text{substitution } \sigma & ::= & \varepsilon \mid \sigma, t/u \mid \sigma, v/z \\
\text{bond } \delta & ::= & \varepsilon \mid \delta, \bar{v}/z \mid \delta, (E :: \bar{t})/u
\end{array}$$

ε is the empty sequence. $\bar{t}$ abbreviates tuple $({}_i t_i)$. Uppercase metavariables (except Γ), e.g. E, stand for relations, e.g. of term tuples $\bar{e}$, usually pairs. Write $rv\ V$ for the reverse of a binary value relation V. Write (ftv) fv for the usual definition of the free (type) variables of a phrase. (dtv) dv stands for the declared (type) variables of a substitution or a base.

Definition 2.1 (Base Formation) Γ *Base*

$$\textit{baseVoid}\ \frac{}{\varepsilon\ \textit{Base}} \qquad \textit{baseAddVal}\ \frac{\Gamma \rhd t\ \textit{Type}}{(\Gamma, z : t)\ \textit{Base}}\, z \notin dv\,\Gamma \qquad \textit{baseAddTyp}\ \frac{\Gamma \rhd t\ \textit{Type}}{(\Gamma, u{<:}t)\ \textit{Base}}\, u \notin dtv\,\Gamma$$

Definition 2.2 (Type Formation [2]) $\Gamma \rhd t\ \mathit{Type}$

$$\mathit{typeVar}\quad \frac{(\Gamma, u{<:}t, \Gamma')\ \mathit{Base}}{\Gamma, u{<:}t, \Gamma' \rhd u\ \mathit{Type}}$$

$$\mathit{typeObjVoid}\quad \frac{\Gamma\ \mathit{Base}}{\Gamma \rhd \mathtt{Ob}(u)\langle\rangle\ \mathit{Type}}\ u \notin \mathit{dtv}\,\Gamma \qquad \mathit{typeObj}\quad \frac{\begin{array}{l}\text{for all } j \in L\\ \quad \Gamma, u{<:}\mathtt{top} \rhd t_j\ \mathit{Type}\\ \quad t_j[u^+]\end{array}}{\Gamma \rhd \mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle\ \mathit{Type}}\ L \neq \emptyset$$

Definition 2.3 (Subtyping) $\Gamma \rhd t{<:}t'$

$$\mathit{subtyVar}\quad \frac{(\Gamma, u{<:}t, \Gamma')\ \mathit{Base}}{\Gamma, u{<:}t, \Gamma' \rhd u{<:}t} \qquad \mathit{subtyRefl}\quad \frac{\Gamma \rhd t\ \mathit{Type}}{\Gamma \rhd t{<:}t} \qquad \mathit{subtyTrans}\quad \frac{\begin{array}{l}\Gamma \rhd t{<:}t'\\ \Gamma \rhd t'{<:}t''\end{array}}{\Gamma \rhd t{<:}t''}$$

$$t \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle$$

$$\mathit{subtyObjVoid}\quad \frac{\Gamma \rhd t\ \mathit{Type}}{\Gamma \rhd t{<:}\mathtt{Ob}(u)\langle\rangle}\ u \notin \mathit{dtv}\,\Gamma$$

$$t_i \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L_i} j\vartheta_{ij} : t_{ij}\rangle\ (i = 1, 2)$$

$$\mathit{subtyObj}\quad \frac{\begin{array}{l}\Gamma \rhd t_2\ \mathit{Type}\\ \text{for all } j \in L_2\\ \quad \Gamma, u{<:}t_1 \rhd \vartheta_{1j}t_{1j}{<:}\vartheta_{2j}t_{2j}\end{array}}{\Gamma \rhd t_1{<:}t_2}\ L_1 \supseteq L_2 \neq \emptyset$$

Definition 2.4 (Subtyping under variances) $\Gamma \rhd \vartheta\ t{<:}\vartheta'\ t'$

$$\mathit{subtyInv}\quad \frac{\Gamma \rhd t\ \mathit{Type}}{\Gamma \rhd \circ t{<:}\circ t} \qquad \mathit{subtyCovar}\quad \frac{\Gamma \rhd t{<:}t'}{\Gamma \rhd \vartheta t{<:}+t'}\ \vartheta \in \{\circ, +\} \qquad \mathit{subtyContrav}\quad \frac{\Gamma \rhd t'{<:}t}{\Gamma \rhd \vartheta t{<:}-t'}\ \vartheta \in \{\circ, -\}$$

Definition 2.5 (Typing)

$$\mathit{termVar}\quad \frac{(\Gamma, z : t, \Gamma')\ \mathit{Base}}{\Gamma, z : t, \Gamma' \rhd z : t} \qquad \mathit{subsum}\quad \frac{\begin{array}{l}\Gamma \rhd e : t\\ \Gamma \rhd t{<:}t'\end{array}}{\Gamma \rhd e : t'}$$

$$t \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle$$

$$\mathit{termObj}\quad \frac{\begin{array}{l}\text{for all } j \in L\\ \quad \Gamma, z_j : t \rhd e_j[t/u] : t_j[t/u]\end{array}}{\Gamma \rhd \mathtt{ob}(u = t)\langle_{j\in L} j = \varsigma(z_j : u)e_j\rangle : t}$$

$$t' \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta'_j : t'_j\rangle$$

$$\mathit{termInvo}\quad \frac{\begin{array}{l}\Gamma \rhd e : t\\ \Gamma \rhd t{<:}t'\end{array}}{\Gamma \rhd e.l : t'_l[t/u]}\ \begin{array}{l}l \in L\\ \vartheta'_l \in \{\circ, +\}\end{array}$$

$$\dfrac{\begin{array}{l} t' \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta'_j : t'_j\rangle \\ termRedef \\ \quad \Gamma \rhd e : t \\ \quad \Gamma \rhd t <: t' \\ \quad \Gamma, u<:t, z : u, z' : u \rhd e' : t'_l \end{array}}{\Gamma \rhd e\langle l := (u<:t, z : u)\varsigma(z' : u)e'\rangle : t} \begin{array}{l} l \in L \\ \vartheta'_l \in \{\circ, -\} \end{array}$$

We consider throughout only phrases and phrase relations well formed according to the rules for that phrase sort (e.g. subtyping rules for types, or typing rules for terms). Stack formation $s : t \xrightarrow{s} t'$ is easily defined from typing as follows.

$$\dfrac{\Gamma, z : t \rhd zs : t'}{\Gamma \rhd s : t \xrightarrow{s} t'} z \notin \text{fv } s$$

Call $\Gamma \rhd t$ a face when $\Gamma \rhd t$. Write $E :: \bar{t}$ when E is a term relation between the types in $\bar{t}$. Val, Tm, Sk are the sets of (well formed) resp. values, terms and stacks. E_{val} abbreviates $E \cap (Val^2)$

Definition 2.6 (Bond formation) *novel notion with formation rules:*

$$\dfrac{}{\varepsilon : \varepsilon} \qquad \dfrac{\begin{array}{l} \delta : \Gamma \\ \Gamma \rhd t \ \mathit{Type} \\ \forall_i \ (v_i : t\delta_i) \end{array}}{(\delta, \bar{v}/z) : (\Gamma, z : t)} z \notin dv\ \delta \qquad \dfrac{\begin{array}{l} \delta : \Gamma \\ \Gamma \rhd t' \ \mathit{Type} \\ E :: \bar{t} \\ \forall_i \ (t_i <: t'\delta_i) \end{array}}{(\delta, (E :: \bar{t})/u) : (\Gamma, u <: t')} u \notin dtv\ \delta$$

We may separate any $\delta : \Gamma$ in its relation substitution δ_0 as well as type and value substitutions δ_i for each i.

The operational semantics of S is given by the next closed relation $\rightsquigarrow$.

Definition 2.7 (Evaluation) *$e \rightsquigarrow v$ of closed term e to a value v is derived by the next rules.*

$$\dfrac{evalVal}{v \rightsquigarrow v} \qquad \dfrac{\begin{array}{l} o \overset{sy}{\equiv} \mathsf{ob}(u = t)\langle_{j\in L} j = \varsigma(z_j : u)e_j\rangle \\ evalInvo \\ \quad e \rightsquigarrow o \\ \quad e_l[t/u, o/z_l] \rightsquigarrow v \end{array}}{e.l \rightsquigarrow v} l \in L$$

$$\dfrac{\begin{array}{l} o \overset{sy}{\equiv} \mathsf{ob}(u = t)\langle_{j\in L} j = \varsigma m_j\rangle \\ o' \overset{sy}{\equiv} \mathsf{ob}(u = t)\langle_{j\in L\setminus\{l\}} j = \varsigma m_j, \\ \quad l = \varsigma(z' : u)e'[o/z]\rangle \\ evalRedef \\ \qquad e \rightsquigarrow o \end{array}}{e\langle l := (u<:t', z : u)\varsigma(z' : u)e'\rangle \rightsquigarrow o'} l \in L$$

Write $\Downarrow$ for the set of evaluating terms e such that $e \Downarrow$, and $\Uparrow$ for nonevaluating terms $e \Uparrow$. Non evaluating well typed closed terms must perpetuate. Abbreviate $E_{\Downarrow^2} \overset{\text{df}}{=} E \cap (\Downarrow^2)$ and $E_{\Uparrow^2} \overset{\text{df}}{=} E \cap (\Uparrow^2)$ and $E_{\Uparrow\Downarrow} \overset{\text{df}}{=} E \cap (\Uparrow \times \Downarrow)$ and $E_{\Downarrow\Uparrow} \overset{\text{df}}{=} E \cap (\Downarrow \times \Uparrow)$. Note that

$$Tm^2 = \Uparrow^2 \cup \Downarrow^2 \cup (\Uparrow \times \Downarrow) \cup (\Downarrow \times \Uparrow) \quad (1)$$
$$E = E_{\Uparrow^2} \cup E_{\Downarrow^2} \cup E_{\Uparrow \times \Downarrow} \cup E_{\Downarrow \times \Uparrow} \quad (2)$$

E is said adequate iff its closed term restriction $E_\varepsilon \subseteq (\Uparrow^2 \cup \Downarrow^2)$; and completely adequate if also $\Uparrow^2 \subseteq E$. E is said subsumptive iff it is closed under the subsumption rule. E is said substitutive iff it is closed under the type and the value substitutivity rules. (These rules appear in respective sections below.) E is said compatible iff it is closed under the compatibility rules of [2].

Operational congruence $\overset{op}{=}_{\Gamma \rhd t}$ is the face indexed family of largest adequate substitutive subsumptive congruence (ie compatible transitive symmetric) binary term relations.

3 Pitts' operators

Define $E \updownarrow S \overset{\text{def}}{\Leftrightarrow} (ES \subseteq (\Uparrow^2 \cup \Downarrow^2)) \Leftrightarrow \forall (\bar{e} \in E, \bar{s} \in S) \bar{e}\bar{s} \in (\Uparrow^2 \cup \Downarrow^2)$. From this and the evaluation definition one can prove the following.

Proposition 3.1 *1. $E_{\Downarrow^2} \updownarrow S$ iff $((rv \leadsto) \circ E_{\Downarrow^2} \circ \leadsto) \updownarrow S$*

2. $E_{\Uparrow\Downarrow} \updownarrow S$ iff $(E_{\Uparrow\Downarrow} \circ \leadsto) \updownarrow S$

3. $E_{\Downarrow\Uparrow} \updownarrow S$ iff $((rv \leadsto) \circ E_{\Downarrow\Uparrow}) \updownarrow S$

4. $E \subseteq E' \updownarrow S' \supseteq S$ implies $E \updownarrow S$

Define $\updownarrow E$ as the largest suitably typed stack relation S such that $E \updownarrow S$ and $\updownarrow S$ as the largest suitably typed term relation E such that $E \updownarrow S$.

Proposition 3.2 *1. $\Uparrow^2 = \updownarrow Sk^2$ and $Sk^2 = \updownarrow (\Uparrow^2)$*

2. $(\Downarrow^2 \cup \Uparrow^2) = \updownarrow \{\bar{\varepsilon}\}$ and $\{\bar{\varepsilon}\} = \updownarrow (\Downarrow^2 \cup \Uparrow^2)$

3. If $X \subseteq X'$ are both either term or stack relations then $\updownarrow X \supseteq \updownarrow X'$

Call $\updownarrow^2 E \overset{\text{df}}{=} \updownarrow (\updownarrow E)$ the Pitts' closure of E; and similarly for stacks $\updownarrow^2 S \overset{\text{df}}{=} \updownarrow (\updownarrow S)$.

A term relation E is said Pitts' closed iff $E = \updownarrow^2 E$ which is tantamount to $E = \updownarrow^2 E'$ or to $E = \updownarrow S$. Similarly S is said Pitts' closed iff $S = \updownarrow^2 S$ iff $S = \updownarrow^2 S'$ iff $S = \updownarrow E$.

Lemma 3.3 *If E is Pitts' closed then* $(\overset{op}{=} \circ E \circ \overset{op}{=}) = E$

The backward inclusion $(\overset{op}{=} \circ E \circ \overset{op}{=}) \supseteq E$ holds trivially by reflexivity of $\overset{op}{=}$ and monotonicity of $\circ$. The forward inclusion may be proved as in lemma 3.14 of [5].

V is said Pitts' value closed when $(\updownarrow^2 V)_{val} = V$. E is said value Pitts closed when $\updownarrow^2 E_{val} = E$. From the lemma and previous propositions we also have the following.

Corollary 3.4

1.

$$\begin{aligned}
\updownarrow^2 E = {\Uparrow^2} \cup \\
&(\rightsquigarrow \circ \overset{op}{=}_{val} \circ (rv \ \rightsquigarrow) \circ E_{\Downarrow^2} \circ \rightsquigarrow \circ \overset{op}{=}_{val} \circ (rv \ \rightsquigarrow)) \cup \\
&(\rightsquigarrow \circ \overset{op}{=}_{val} \circ (rv \ \rightsquigarrow) \circ E_{\Downarrow\Uparrow} \circ \overset{op}{=}) \cup \\
&(\overset{op}{=} \circ E_{\Uparrow\Downarrow} \circ \rightsquigarrow \circ \overset{op}{=}_{val} \circ (rv \ \rightsquigarrow))
\end{aligned}$$

2. $\updownarrow^2 V = \updownarrow^{\rightsquigarrow}_{\rightsquigarrow} (\overset{op}{=} \circ V \circ \overset{op}{=})$.

3. E is value Pitts closed iff E_{val} is Pitts value closed.

4. V is Pitts value closed iff $\updownarrow^2 V$ is value Pitts closed.

5. If V is Pitts' value closed then $V = (\overset{op}{=}_{val} \circ V \circ \overset{op}{=}_{val})$.

6. If E is value Pitts' closed then $E = \updownarrow^{\rightsquigarrow}_{\rightsquigarrow} (\overset{op}{=}_{val} \circ E_{val} \circ \overset{op}{=}_{val})$.

4 Open bisimilarity from relators

Let $t \overset{\text{df}}{=} \mathtt{Ob}(u)\langle_{j \in L} j \vartheta_j : t_j \rangle$ in

$$\frac{\forall (j \in L_{t+})({}_i v_i . j) \in \mathcal{B}_{t_j}(\delta, E/u) :: ({}_i t_j \delta_i)}{\bar{v} \in \mathtt{obi}_{t,\delta,u}\ E :: ({}_i t \delta_i)}$$

$$\frac{\forall (j \in L_{t-}) \forall (r : t \xrightarrow{s} t' \lhd \Gamma)({}_i v_i r \delta_i) \in \updownarrow^2 E}{\bar{v} \in \mathtt{obr}_{t,\delta,\Gamma}\ E :: ({}_i t \delta_i)}$$

$$\mathtt{ob}_{t,\delta,\Gamma,u}(E) \overset{\text{df}}{=} (\mathtt{obi}_{t,\delta,u}\ E) \cap (\mathtt{obr}_{t,\delta,\Gamma}\ E) :: ({}_i t \delta_i)$$

The obr relator experiments with bisimilar instantiations $({}_i r \delta_i)$ of an open redefinition frame r, thus with same outmost constructor; not just identical closed frames $({}_i r)$ as in [2].

Since evaluation is call-by-value as in [3], we need one relator for each value constructor of the language.

Definition 4.1 (open type indexed relators) *Assuming $\Gamma \rhd t$ and $\delta : \Gamma$, the open type indexed family of relators $\mathcal{V}_t$ and $\mathcal{B}_t$ over δ are defined as follows.*

$$\begin{aligned}
\mathcal{B}_t\ \delta &\overset{\text{df}}{=} \updownarrow^2 (\mathcal{V}_t\ \delta)\\
\mathcal{V}_u\ \delta &\overset{\text{df}}{=} (\delta_0\ u)_{val} :: ({}_i\delta_i\ u)\\
\mathcal{V}_{\mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle}\ \delta &\overset{\text{df}}{=} \nu(X)\mathtt{ob}_{t,\delta,\Gamma,u}\ X :: ({}_i\mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\delta_i\rangle))
\end{aligned}$$

Notice the maximum fixed point for each object type constructor. Note that $\mathcal{B}_t\ \delta :: ({}_it\delta_i)$ and $(\mathcal{V}_t\ \delta) :: ({}_it\delta_i)$.

Definition 4.2 (Bisimilar bond)

$$\frac{}{\varepsilon \updownarrow \varepsilon} \qquad \frac{\begin{array}{l}\delta \updownarrow \Gamma\\ \bar{v} \in (\mathcal{B}_t\ \delta)\\ (\delta, \bar{v}/z) : (\Gamma, z : t)\end{array}}{(\delta, \bar{v}/z) \updownarrow (\Gamma, z : t)} \qquad \frac{\begin{array}{l}\delta \updownarrow \Gamma\\ E \subseteq (\mathcal{B}_{t'}\ \delta|\bar{t})\\ (\delta, (E :: \bar{t})/u) : (\Gamma, u <: t')\end{array}}{(\delta, (E :: \bar{t})/u) \updownarrow (\Gamma, u <: t')}$$

Lemma 4.3 *If $\delta \updownarrow \Gamma$ and $\Gamma \rhd t$ Type then $(\mathcal{B}_t\ \delta)_{val} = (\updownarrow^2 (\mathcal{V}_t\ \delta))_{val} = \mathcal{V}_t\ \delta$.*

Definition 4.4 (Bisimilarity) *$\mathcal{V}$ and $\mathcal{B}$ is the face indexed family $\mathcal{B}_{\Gamma \rhd t}$ of open term binary relations defined as follows:*

$$\frac{\begin{array}{l}\forall(\delta \updownarrow \Gamma)\\ \quad ({}_iv_i\delta_i) \in \mathcal{V}_t\ \delta\end{array}}{\bar{v} \in \mathcal{V}_{\Gamma \rhd t}} \qquad \frac{\begin{array}{l}\forall(\delta \updownarrow \Gamma)\\ \quad ({}_ie_i\delta_i) \in \mathcal{B}_t\ \delta\end{array}}{\bar{e} \in \mathcal{B}_{\Gamma \rhd t}}$$

Note that $\mathcal{V}_{\Gamma \rhd t} = (\mathcal{B}_{\Gamma \rhd t})_{val}$

5 Adequacy

Lemma 5.1 *$\mathcal{B}$ is adequate.*

Proof: Consider any $\bar{e} \in \mathcal{B}_{\Gamma \rhd t}$. Then $\bar{e} \in (\mathcal{B}_{\Gamma \rhd t})_\varepsilon$ iff for every i we have $\rhd e_i : t$ and for all $\delta \updownarrow \Gamma$, $\bar{e}\delta = \bar{e} \in (\mathcal{B}_t\ \delta) = \mathcal{B}_t\ \varepsilon$. Thus $(\mathcal{B}_{\Gamma \rhd t})_\varepsilon = \mathcal{B}_{\varepsilon \rhd t}$. If $(\text{ftv}\ t) \neq \{\}$ then $\mathcal{B}_{\varepsilon \rhd t} = \{\}$ which is trivially adequate. If $\rhd t$ Type then by definition $\mathcal{B}_t\ \varepsilon = \updownarrow^2 V$ for some $V :: ({}_it)$. As $V \updownarrow (\varepsilon, \varepsilon)$, that is $(\varepsilon, \varepsilon) \in \updownarrow V$, then by definition $(\updownarrow^2 V) \updownarrow (\varepsilon, \varepsilon)$; so $\mathcal{B}_t\ \varepsilon \subseteq (\Uparrow^2 \cup \Downarrow^2)$. □

6 Subsumptiveness

E is said subsumptive iff it is closed under type subsumption.

$$\frac{\begin{array}{l}\bar{e} \in E_{\Gamma \rhd t}\\ \Gamma \rhd t <: t'\end{array}}{\bar{e} \in E_{\Gamma \rhd t'}}$$

Write $(E :: \bar{t}) <:: (E' :: \bar{t}')$ iff (1) $t_i <: t'_i$ for every i and (2) $E :: \bar{t}$ and (3) $E' :: \bar{t}'$ and (4) $E \subseteq E'$.

Lemma 6.1 (Subsumption lemma) *If* $\Gamma \rhd t <: t'$ *then* $\forall(\delta \updownarrow \Gamma)(\mathcal{B}_t\ \delta :: t\delta) <:: (\mathcal{B}_{t'}\ \delta :: t'\delta)$

(1) Proved by induction on the subtyping derivation, as $\delta \updownarrow \Gamma$ implies $\delta_i : \Gamma$ for every i. (2) and (3) hold by the definition of $\mathcal{B}_t\ \delta$. (4) follows by the monotonicity of $\updownarrow^2$ from $\mathcal{V}_t\ \delta \subseteq \mathcal{V}_{t'}\ \delta$ which comes from $(\nu\ \mathsf{ob}_{t,\delta,\Gamma,u}) \subseteq \mathsf{ob}_{t',\delta,\Gamma,u}(\nu\ \mathsf{ob}_{t,\delta,\Gamma,u})$. This holds because $\rhd t\delta_i <: t'\delta_i$ implies $(\mathsf{ob}_{t,\delta,\Gamma,u}\ X) \subseteq (\mathsf{ob}_{t',\delta,\Gamma,u}\ X)$.

Proposition 6.2 *$\mathcal{B}$ is subsumptive.*

Proof: Goal: $\bar{e} \in \mathcal{B}_{\Gamma \rhd t'}$. Assume (H1) $\bar{e} \in \mathcal{B}_{\Gamma \rhd t}$ and (H2) $\Gamma \rhd t <: t'$. By the subsumption lemma, the latter implies $\forall(\delta \updownarrow \Gamma)(\mathcal{B}_t\ \delta) \subseteq (\mathcal{B}_{t'}\ \delta)$. As (H1) stands for $\forall(\delta \updownarrow \Gamma)({}_ie_i\delta_i) \in \mathcal{B}_t\ \delta$, then $\forall(\delta \updownarrow \Gamma)({}_ie_i\delta_i) \in \mathcal{B}_{t'}\ \delta$ which is nothing but the goal. □

7 Substitutivity

A term relation E is said substitutive iff $E[I_{ty} \cup (E)_{val}/] \subseteq E$; that is when closed under the next rules

$$\frac{\bar{e} \in E_{\Gamma, u <: t', \Gamma' \rhd t''} \qquad \Gamma \rhd t <: t'}{({}_ie_i[t/u]) \in E[t/u]_{\Gamma, \Gamma'[t/u] \rhd t''[t/u]}} \qquad \frac{\bar{e} \in E_{\Gamma, z:t, \Gamma' \rhd t'} \qquad \bar{v} \in V_{\Gamma \rhd t}}{({}_ie_i[v_i/z]) \in E[V/z : t]_{\Gamma, \Gamma' \rhd t'}}$$

Lemma 7.1 (Type substitution lemma) *If* $(\delta, (\mathcal{B}_t\ \delta :: ({}_it\delta_i))/u, \delta') : (\Gamma, u <: t', \Gamma')$ *and* $\Gamma, u <: t', \Gamma' \rhd e_i : t''$ *and* $\Gamma \rhd t <: t'$ *then* $({}_ie_i(\delta_i, t\delta_i/u, \delta'_i)) \in \mathcal{B}_{t''}(\delta, \mathcal{B}_t\ \delta/u, \delta')$ *iff* $({}_ie_i[t/u](\delta_i, \delta'_i)) \in \mathcal{B}_{t''[t/u]}(\delta, \delta')$

Provable by induction on the structure of term e_i.

Lemma 7.2 (Value substitution lemma) *If* $\Gamma, z : t, \Gamma' \rhd e_i : t'$ *and* $\Gamma \rhd v_i : t$ *and* $\delta : \Gamma$ *and* $\delta' : \Gamma'$ *then* $({}_ie_i[v_i/z](\delta, \delta')) = ({}_ie_i(\delta_i, v_i\delta_i/z, \delta'_i)) \in \mathcal{B}_{t'}(\delta, \delta')$

Provable by induction on the structure of term e_i.

Proposition 7.3 *$\mathcal{B}$ is substitutive.*

Proof:

Closure under type substitution Goal: $({}_i e_i[t/u]) \in \mathcal{B}_{\Gamma,\Gamma'[t/u] \triangleright t''[t/u]}$. Assume $H \overset{\text{df}}{=} \bar{e} \in \mathcal{B}_{\Gamma,u<:t',\Gamma',\Gamma' \triangleright t''}$ and $H' \overset{\text{df}}{=} \Gamma \triangleright t <: t'$. From H', the subsumption lemma and given any $\delta \updownarrow \Gamma$ we have $(\mathcal{B}_t\ \delta) \subseteq (\mathcal{B}_{t'}\ \delta)$. Then, from H, $\forall((\delta, \mathcal{B}_t\ \delta/u, \delta') \updownarrow (\Gamma, u <: t', \Gamma'))({}_i e_i(\delta_i, t\delta_i/u, \delta'_i) \in \mathcal{B}_{t''}(\delta, \mathcal{B}_t\ \delta/u, \delta')$. By the type substitution lemma, this is tantamount to $\forall((\delta,\delta') \updownarrow (\Gamma, \Gamma'[t/u]))({}_i e_i[t/u](\delta_i, \delta'_i)) \in \mathcal{B}_{t''[t/u]}(\delta,\delta')$ which is what the goal stands for.

Closure under value substitution Goal: $({}_i e[v_i/z]) \in \mathcal{B}_{\Gamma,\Gamma' \triangleright t'}$. Assume $H \overset{\text{df}}{=} \bar{e} \in \mathcal{B}_{\Gamma,z:t,\Gamma' \triangleright t'}$ and $H' \overset{\text{df}}{=} \bar{v} \in \mathcal{B}_{\Gamma \triangleright t}$. The latter means $\forall(\delta \updownarrow \Gamma)({}_i v_i \delta_i) \in \mathcal{B}_t\ \delta$. By this and H one has $\forall((\delta, ({}_i v_i \delta_i)/z, \delta') \updownarrow (\Gamma, z : t, \Gamma'))({}_i e_i(\delta_i, v_i\delta_i/z, \delta'_i)) \in \mathcal{B}_{t'}(\delta,\delta')$. This is equivalent by the value substitution lemma to $\forall(\delta,\delta') \updownarrow (\Gamma,\Gamma')({}_i e_i[v_i/z](\delta_i,\delta'_i)) \in \mathcal{B}_{t'}(\delta,\delta')$ which is the definition of the goal.

□

8 Compatibility

A term relation is said compatible iff it is closed under the next compatibility rules.

$$\textit{compatVar}\quad \frac{(\Gamma, z : t, \Gamma')\ \text{Base}}{({}_i z) \in \mathcal{B}_{\Gamma,z:t,\Gamma' \triangleright z:t}}$$

$$\begin{array}{l} t \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle \\ o_i \overset{sy}{\equiv} \mathtt{ob}(u = t)\langle_{j\in L} j = \varsigma(z_j : u)e_{ij}\rangle \\ \textit{compatObj} \end{array} \quad \frac{\forall(j \in L)\quad ({}_i e_{ij}[t/u]) \in \mathcal{B}_{\Gamma,z_j:t \triangleright t_j[t/u]}}{({}_i o_i) \in \mathcal{B}_{\Gamma \triangleright t}}$$

$$\begin{array}{l} t' \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta'_j : t'_j\rangle \\ \textit{compatInvo} \end{array} \quad \frac{\begin{array}{c}\bar{e} \in \mathcal{B}_{\Gamma \triangleright t} \\ \Gamma \triangleright t<:t'\end{array}}{({}_i e_i.l) \in \mathcal{B}_{\Gamma \triangleright t'_l[t/u]}} \quad \begin{array}{l} l \in L \\ \vartheta'_l \in \{\circ, +\}\end{array}$$

$$\begin{array}{l} t' \overset{sy}{\equiv} \mathtt{Ob}(u)\langle_{j\in L} j\vartheta'_j : t'_j\rangle \\ r_i \overset{sy}{\equiv} \langle l{:=}(u<:t, z : u)\varsigma\varsigma(z' : u)e'_i\rangle \\ \textit{compatRedef} \end{array} \quad \frac{\begin{array}{l}\bar{e} \in \mathcal{B}_{\Gamma \triangleright t} \\ \Gamma \triangleright t<:t' \\ \bar{e'} \in \mathcal{B}_{\Gamma,u<:t,z:u,z':u \triangleright t'_l}\end{array}}{({}_i e_i r_i) \in \mathcal{B}_{\Gamma \triangleright t}} \quad \begin{array}{l} l \in L \\ \vartheta'_l \in \{\circ, -\}\end{array}$$

Proposition 8.1 *$\mathcal{B}$ is compatible.*

Proof: I show the full backward proof tree just once for the easier case of variable compatibility. In the remaining cases I only hint on non routine proof aspects.

variable Let $H \overset{\text{df}}{=} (\Gamma, z : t, \Gamma')$ Base

1. $({}_i z) \in \mathcal{B}_{\Gamma,z:t,\Gamma' \triangleright t} \dashv H$
   $\forall(\delta, \bar{v}/z, \delta') \updownarrow (\Gamma, z : t, \Gamma')({}_i z[\delta_i, v_i/z, \delta'_i]) \in \mathcal{B}_t\ \delta \dashv H$
   $({}_i z[v_i/z]) \in \mathcal{B}_t\ \delta \dashv H, (\delta, \bar{v}/z, \delta') \updownarrow (\Gamma, z : t, \Gamma')$
   $\bar{v} \in \mathcal{B}_t\ \delta \dashv H, \delta \updownarrow \Gamma, \bar{v} \in \mathcal{B}_t\ \delta, \delta' \updownarrow \Gamma'$

invocation provable using the subsumption lemma and routine applications of other previous lemmas.

object the hard aspect is to find a bisimulation relation $O \subseteq \mathsf{ob}_{t,\delta,\Gamma}\ O$ such that $({}_i o_i \delta_i) \in O$.

redefinition key in the proof is a methodwise object bisimilarity relation $\overset{m}{=}_{t,\delta}$ defined as follows.

$$\begin{array}{rcl} t & \overset{\text{df}}{=} & \mathsf{Ob}(u)\langle_{j\in L} j\vartheta_j : t_j\rangle \\ o'_i & \overset{\text{df}}{=} & \mathsf{ob}(u = t\delta_i)\langle_{j\in L} j = \varsigma m'_{ij}\rangle \\ \bar{o}' \in \overset{m}{=}_{t,\delta} & \overset{\text{def}}{\Leftrightarrow} & \forall(j \in L)({}_i\lambda\ m'_{ij}[t\delta_i/u]) \in \lambda(\mathcal{B}_t\ \delta, \mathcal{B}_{t_j[t/u]}\ \delta) \end{array}$$

□

9 Transitivity and Symmetry

Note that adequacy is an equivalence and that the subsumption, compatibility, type and value substitution rules preserve transitivity and symmetry. Transitivity and symmetry are provable directly as hinted next.

9.1 Transitivity

Lemma 9.1 $(\mathcal{V}_t\ \delta_0) \circ (\mathcal{V}_t\ \delta'_0) \subseteq \mathcal{V}_t\ (\delta_0 \circ \delta'_0)$
$(\mathcal{B}_t\ \delta_0) \circ (\mathcal{B}_t\ \delta'_0) \subseteq \mathcal{B}_t\ (\delta_0 \circ \delta'_0)$

Lemma 9.2 $\forall(\delta \updownarrow \Gamma)\exists(\delta', \delta'' \updownarrow \Gamma)\delta_0 = (\delta'_0 \circ \delta''_0) \wedge \delta'_1 = \delta_1 \wedge \delta'_2 = \delta''_1 \wedge \delta''_2 = \delta_2$

Proposition 9.3 *$\mathcal{B}$ is transitive.*

This may be proved from the preceeding lemmas.

9.2 Symmetry

Lemma 9.4 $(rv\ \delta') \updownarrow \Gamma$ *whenever* $\delta' \updownarrow \Gamma$

Lemma 9.5 $rv(\updownarrow^2\ E) = \updownarrow^2 (rv\ E)$

Lemma 9.6 $rv\ (\mathcal{V}_t\ \delta) = \mathcal{V}_t(rv\ \delta)$

Lemma 9.7 *$\mathcal{B}$ is symmetric.*

This may be proved from the preceeding lemmas.

10 Operational congruence equals bisimilarity

Theorem 10.1 $\overset{op}{=}_{\Gamma \rhd t} = \mathcal{B}_{\Gamma \rhd t}$

Proof:

Right to left inclusion ($\overset{op}{=}_{\Gamma \rhd t} \supseteq \mathcal{B}_{\Gamma \rhd t}$) $\mathcal{B}_{\Gamma \rhd t}$ is adequate, subsumptive, substitutive, compatible, transitive and symmetric, as proved above. $\overset{op}{=}_{\Gamma \rhd t}$ is by definition the family of largest relations with precisely such properties. So the inclusion holds.

Left to right inclusion ($\overset{op}{=}_{\Gamma \rhd t} \subseteq \mathcal{B}_{\Gamma \rhd t}$) Assume any $({}_i e_i) \in \overset{op}{=}_{\Gamma \rhd t}$. Given any δ such that $\delta \updownarrow \Gamma$ we want to prove $({}_i e_i \delta_i) \in \mathcal{B}_t\ \delta$. Note that $(e_1 \delta_1, e_1 \delta_1) \in \overset{op}{=}_{t\delta_1}$ as this is reflexive since it is compatible by definition. Also $(e_1 \delta_1, e_1 \delta_2) \in (\mathcal{B}_t\ \delta)$ which follows from $\delta \updownarrow \Gamma$ by induction on the structure of e_1.And $(e_1 \delta_2, e_2 \delta_2) \in \overset{op}{=}_{t\delta_2}$ by substitutivity of $\overset{op}{=}$. Thus $({}_i e_i \delta_i) \in (\overset{op}{=}_{t\delta_1} \circ \mathcal{B}_t\ \delta \circ \overset{op}{=}_{t\delta_2})$. As $\mathcal{B}_t\ \delta = \updownarrow^2 \mathcal{V}_t\ \delta$ is trivially Pitts closed, the inclusion holds by lemma 3.3.

□

11 Further work

I conjecture that the extensionality result also holds for S_{cbv} and $S_{\forall}$.

As for the relation between bisimilarity and contextual congruence, I conjecture that: they coincide for S_{cbv} with only closed types (as for full S_{λ}); but that for S bisimilarity is fined grained than contextual congruence (as discussed by Gordon for $S_{\forall}$).

I have been generalizing the calculus theory for preorders (dropping symmetry), roughly replacing symmetric notions and notation by preorder counterparts, for instance: adequate by preadequate, closure by preclosure, etc.

I have investigated a first-order mu logic for S_{λ}, distilled years ago from case studies on specification and verification of object oriented programs. I am considering alternative semantics, developing proof systems and analysing practically relevant meta theory, namely soundness. A higher-level logic over such (positive) core logic is also envisaged.

Finally I intend to identify object specification and verification techniques and patterns, explore semantical relators for specification (generalizing the grammatical ones already used) and revisit such practice as application of the developed theory.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[2] A. Gordon. Operational equivalences for untyped and polymorphic object calculi. In A. Pitts and A. Gordon, editors, *Higher-Order Operational Techniques in Semantics.* Cambridge University Press, 1997.

[3] A. M. Pitts. Existential types: Logical relations and operational equivalence. Number 1443. Springer-Verlag, 1998.

[4] A. M. Pitts. Parametric polymorphism and operational equivalence. *Electronic Notes in Theoretical Computer Science*, 10, 1998.

[5] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10, 2000.

Coproducts of Ideal Monads (Extended Abstract)

Neil Ghani
Dept. of Math. and Comp. Sci.
University of Leicester
University Road
Leicester LE1 7RH, UK
ng13@mcs.le.ac.uk

Tarmo Uustalu
Institute of Cybernetics
Tallinn Technical University
Akadeemia tee 21
EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

Abstract

The development of a calculus of monad combinators has been a subject of much recent research. Although a general construction exists, its generality is reflected in its complexity which limits the applicability of this construction. Following our own research [12], and that of Hyland, Plotkin and Power [8], we are looking for specific situations when simpler constructions are available. This paper uses fixed points to give a simple construction of the coproduct of two *ideal monads*.

A Brief Reminder on Monads A *monad* $\mathsf{T} = \langle T, \eta, m\rangle$ on a category $\mathcal{C}$ is given by an endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$, called the *action*, and two natural transformations, $\eta : 1 \longrightarrow T$, called the *unit*, and $m : TT \longrightarrow T$, called the *multiplication* of the monad, satisfying the *monad laws*: $m \cdot T\eta = T = m \cdot \eta T$, and $m \cdot T = m \cdot mT$. We write m for the multiplication rather than the usual μ since we reserve μ for least fixed points and free monads.

The canonical example of monads is that of term algebras. Every signature Σ defines a monad $\mathsf{T}_\Sigma : \mathbf{Set} \longrightarrow \mathbf{Set}$ whose action maps a set to the term algebra over this set. The unit maps a variable to the associated term, while the multiplication describes the process of substitution. The monad laws ensure that substitution behaves correctly, i.e. substitution is associative and the variables are left and right units. Monads also model a number of other important structures in computer science, such as (many-sorted) algebraic theories, non-well-founded syntax [15, 1, 6], term graphs [7], calculi with variable binders [5], term rewriting systems [11], and, via computational monads [13], state-based computations, exceptions, continuations etc. These applications involve base categories other than **Set** and the desire for a uniform treatment underpins their monadic axiomatisation.

Combining Monads A prerequisite for modular reasoning is an understanding of how individual components of a large system interact with each other. In particular, if different components of a system are modelled by different monads, how do we combine these monads to represent the overall system. Concretely, if two term rewriting systems R and R' are modelled by monads T_R and $\mathsf{T}_{R'}$ how can we reason about the combined system $R + R'$ via its representing monad $\mathsf{T}_{R+R'}$? Alternatively, given a monad modelling exceptions and a monad modelling state transformations, can we derive a monad modelling computations which can either raise exceptions or modify the state? One possible answer to these questions is given by the theory of *monad transformers* [14]. Although the concept of a monad transformer is rather elegant, in our opinion the definition is too general to support an adequate meta-theory. For example, given a monad it is not clear whether it is possible and, if so, how to define an associated monad transformer.

This paper is based upon the thesis that colimits of monads provide an appropriate framework for combining monads. A construction of the colimit of monads was given by Kelly [9] but the generality of the construction is reflected in its complexity which can be deterring even for experienced category theorists and which certainly limits its applicability. Consequently, recent research has focussed on i) *coproducts* of monads which model combinations of systems where there is no sharing as in the examples above; ii) providing alternative constructions which, by restricting to special cases, are significantly simpler and hence easier to apply in practice; and iii) although the existence of the coproduct of two monads usually follows from general categorical considerations, it is often unclear what the action of this monad is. Hence we seek alternative functorial and fixed point characterisations of the coproduct of monads which make explicit the action.

The rest of this paper recalls what is known for free monads and then tackles the question of composing *ideal monads*. Our results are similar to those in [12] but our proofs are much simpler and hence more useful in practice. This is because we have used fixed points to hide the construction of various cocones etc. Given a functor $F : \mathcal{C} \longrightarrow \mathcal{C}$, we denote (the carrier of) its initial algebra by μF.

Coproducts of Free Monads Monad morphisms between monads T and H are natural transformations $h : T \longrightarrow H$ which preserve the unit and multiplication of the monads. Given a fixed base category $\mathcal{C}$, monads and monad morphisms on $\mathcal{C}$ form a category $\mathbf{Mon}(\mathcal{C})$. To achieve abstraction we follow the standard practice of replacing a signature Σ with the associated polynomial functor $F_\Sigma : \mathbf{Set} \to \mathbf{Set}$. Given an endofunctor $F : \mathcal{C} \longrightarrow \mathcal{C}$, the free monad on F is written F^μ and is defined as the universal arrow from F to the forgetful functor $U : \mathbf{Mon}(\mathcal{C}) \longrightarrow [\mathcal{C},\mathcal{C}]$. The first important connection between fixed points and monads is:

Proposition 1 [4]: Given a functor $F : \mathcal{C} \longrightarrow \mathcal{C}$, the free monad is the initial $1 + F \circ \_ : [\mathcal{C},\mathcal{C}] \longrightarrow [\mathcal{C},\mathcal{C}]$ algebra.

Note that the term algebra monad is such a free monad. We could formalise free monads as left adjoint to the forgetful functor by using *lfp*-categories and *finitariness* [10, 3] but in this paper we want to work without such technical assumptions. There are a number of other simple connections between fixed points and monads. For example, the free completely iterative monad [1] arises as the final $1+F\circ\_ : [\mathcal{C},\mathcal{C}] \longrightarrow [\mathcal{C},\mathcal{C}]$ coalgebra while the term graph monad and rational monads [7, 2] are also $1 + F \circ \_$ fixed points. Coproducts of free monads are easy to construct and understand.

Proposition 2: Let F and G be functors. Then $F^\mu + G^\mu = (F+G)^\mu = \mu(1 + F\circ\_ + G\circ\_)$.

In general, this proposition indicates the sort of analysis we want, that is a reduction of the construction of the coproduct monads to a fixed point formula involving endofunctors. A more general result [8] shows that if S is any monad and F^μ a free monad, then $S + F^\mu = S(FS)^\mu$. This is a significant improvement as it reduces the coproduct of any monad with a free monad to functorial composition. Furthermore, this functorial formula can be reduced to a fixed point formula: $S(FS)^\mu = S(\mu(1 + FS\circ\_)) = \mu(S(1 + F\circ\_))$. The last equality is an application of the *rolling lemma* for fixed points.

Coproducts of Ideal Monads The core of this abstract is the use of fixed points to calculate the coproduct of a large variety of monads—the so-called *ideal monads* [1]. These were introduced to describe those monads which can be decomposed into their variable and non-variable parts.

Formally, a monad $\langle T, \eta, m\rangle$ is ideal iff there is a functor T_0 such that $T = 1 + T_0$, the unit is the left injection and there is a natural transformation $m_0 : T_0T \longrightarrow T_0$ such that

$$\begin{tikzcd}
T_0T \arrow[r, "in_2T"] \arrow[d, "m_0"'] & TT \arrow[d, "m"] \\
T_0 \arrow[r, "in_2"'] & T
\end{tikzcd}$$

We write ideal monads in the form $1 + T_0$ for simplicity and leave the restricted form of multiplication m_0 implicit. A monad morphism $f : 1 + T_0 \longrightarrow R$ whose source is an ideal monad has its action on 1 forced by the monad laws and is hence of the form $[\eta^R, f_0]$ where $f_0 : T_0 \longrightarrow R$. Examples of ideal monads include free monads, free completely iterative monads etc. The fundamental observation behind the construction of the coproduct $R + S$ of ideal monads $R = 1 + R_0$ and $S = 1 + S_0$ is that i) $R + S$ should contain as submonads R and S; and ii) $R + S$ should be closed under the application of R_0 and S_0. Hence $R + S$ should consist of alternating sequences beginning from R_0 or S_0. Thus we ask for least fixed points

$$T_1 \cong R_0(1 + T_2) \qquad\qquad T_2 \cong S_0(1 + T_1)$$

and write t_1, t_2 for the structure maps. Intuitively T_1 consists of elements in $R+S$ whose top layer is a non-variable R-layer (captured by the use of R_0) and whose next layers are either variables or a non-variable S layer etc. We henceforth assume T_1 and T_2 exist, for example, we may require C to have ω-colimits and for R_0 and S_0 preserve them.

Proposition 3: The action of the coproduct of ideal monads $1 + R_0$ and $1 + S_0$ is the functor $T = 1 + (T_1 + T_2)$.

Functoriality of T is obvious. The unit η is the injection $1 \xrightarrow{in_1} 1 + (T_1 + T_2) = T$. The multiplication m is $[T, in_2 \cdot (m_1 + m_2)]$ where $m_1 : T_1T \longrightarrow T_1$ and $m_2 : T_2T \longrightarrow T_2$ can be constructed by generalized mutual iteration:

$$\begin{tikzcd}
R_0(1+T_2)T \arrow[r, "t_1T"] \arrow[d, "R_0(T+m_2)"'] & T_1T \arrow[d, "m_1"] & T_2T \arrow[d, "m_2"'] & S_0(1+T_1)T \arrow[l, "t_2T"'] \arrow[d, "S_0(T+m_1)"] \\
R_0(T+T_2) \arrow[r, "p_1"'] & T_1 & T_2 & S_0(T+T_1) \arrow[l, "p_2"]
\end{tikzcd}$$

where p_1, p_2 denote the composites

$$R_0(T + T_2) \longrightarrow R_0((1 + T_2) + T_1) \xrightarrow{R_0((1+T_2)+t_1^{-1})} R_0R(1 + T_2) \xrightarrow{m_0^R(1+T_2)} R_0(1 + T_2) \xrightarrow{t_1} T_1$$
$$S_0(T + T_1) \longrightarrow S_0((1 + T_1) + T_2) \xrightarrow{S_0((1+T_1)+t_2^{-1})} S_0S(1 + T_1) \xrightarrow{m_0^S(1+T_1)} S_0(1 + T_1) \xrightarrow{t_1} T_1$$

The unit laws are satisfied trivially. For the associativity of multiplication one explicitly constructs $m_1^{(3)} = T_1TT \longrightarrow T_1$ and $m_2^{(3)} = T_2TT \longrightarrow T_2$ by generalized mutual iteration to then show that both $m \cdot Tm$ and $m \cdot mT$ equal $m^{(3)} = [m, in_2 \cdot (m_1^{(3)} + m_2^{(3)})] : TTT \longrightarrow T$.

Next, we need monad morphisms $R, S \longrightarrow T$ to play the role of injections. They are given by the composites

$$R_0 \xrightarrow{R_0 in_1} R_0(1 + T_2) \xrightarrow{t_1} T_1 \qquad\qquad S_0 \xrightarrow{S_0 in_1} S_0(1 + T_1) \xrightarrow{t_2} T_2$$

These maps are natural as we work in a functor category, preservation of the unit is again trivial while preservation of the multiplication is a short diagram chase.

Finally, we turn to the construction of copairing. Given monad morphisms $f, g : R, S \longrightarrow H$ induced by $f_0, g_0 : R_0, S_0 \longrightarrow H$ we construct maps $h_1 : T_1 \longrightarrow H$ and $h_2 : T_2 \longrightarrow H$ by mutual iteration:

$$\begin{tikzcd}
R_0(1+T_2) \arrow[r, "t_1"] \arrow[d, "R_0(1+h_2)"'] & T_1 \arrow[d, "h_1"] & T_2 \arrow[d, "h_2"'] & S_0(1+T_1) \arrow[l, "t_2"'] \arrow[d, "S_0(1+h_1)"] \\
R_0(1+H) \arrow[r, "q_1"'] & H & H & S_0(1+H) \arrow[l, "q_2"]
\end{tikzcd}$$

where q_1, q_2 are defined as the composites

$$R_0(1+H) \xrightarrow{R_0[\eta^H, H]} R_0H \xrightarrow{f_0H} HH \xrightarrow{m^H} H \qquad S_0(1+H) \xrightarrow{S_0[\eta^H, H]} S_0H \xrightarrow{g_0H} HH \xrightarrow{m^H} H$$

By diagram chasing one shows that $[\eta^H, [h_1, h_2]]$ is a monad morphism, that it is a mediating morphism and that it is the unique such. Intuitively, uniqueness should not be surprising since any other monad morphism $h : T \longrightarrow H$ must equal $[\eta^H, [h_1, h_2]]$ i) on variables because of the laws on monad morphisms; ii) on R_0 and S_0 because they both form cones; and iii) on all other elements of T since they are essentially multiplications of R_0 and S_0 which are preserved by monad morphisms.

Acknowledgements The first author's research was supported by EPSRC under grant No. GR/M96230/01 *Categorical Rewriting: Monads and Modularity.* The second author's research was supported by the Estonian Science Foundation under grant No. 5567 and his participation at ETAPS 2003 was made possible by a travel grant from the Estonian Information Technology Foundation.

References

[1] P. Aczel, J. Adámek, and J. Velebil. A coalgebraic view of infinite trees and iteration. In A. Corradini, M. Lenisa, and U. Montanari, editors, *Proc. of CMCS'01*, volume 44(1) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2001.

[2] J. Adámek, S. Milius, and J. Velebil. Free iterative theories: a coalgebraic view. Accepted for publication in *Mathematical Structures in Computer Science*, 2002.

[3] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*, volume 189 of *London Mathematical Society Lecture Notes.* Cambridge University Press, 1994.

[4] M. Barr. Coequalizers and free triples. *Math. Z.*, 116:307–322, 1970.

[5] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of LICS'99*, pages 193–202. IEEE CS Press, 1999.

[6] N. Ghani, C. Lüth, F. de Marchi, and J. Power. Algebras, coalgebras, monads and comonads. In A. Corradini, M. Lenisa, and U. Montanari, editors, *Proc. of CMCS'01*, volume 44(1) of *Electronic Notes in Theoretical Computer Science.* 2001.

[7] N. Ghani, C. Lüth, and F. De Marchi. Coalgebraic monads. In L. S. Moss, editor, *Proc. of CMCS'02*, volume 65(1) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2002.

[8] M. Hyland, G. Plotkin, and J. Power. Combining computational effects: Commutativity and sum. In A. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Proc. of IFIP 17th World Computer Congress, TC1 Stream / TCS 2002*, volume 223 of *IFIP Conference Proceedings*, pages 474–484. Kluwer Academic Publishers, 2002.

[9] G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bull. of Australian Mathematical Society*, 22:1–83, 1980.

[10] G. M. Kelly and J. Power. Adjunctions whose counits are equalizers, and presentations of finitary monads. *Journal of Pure and Applied Algebra*, 89:163–179, 1993.

[11] C. Lüth. *Categorical Term Rewriting: Monads and Modularity.* PhD thesis, University of Edinburgh, 1998.

[12] C. Lüth and N. Ghani. Monads and modularity. In A. Armando, editor, *Proc. of FroCoS'02*, number 2309 in Lecture Notes in Computer Science, pages 18–32. Springer Verlag, 2002.

[13] E. Moggi. Computational lambda-calculus and monads. In *Proc. of LICS'89*, pages 14–23. IEEE CS Press, 1989.

[14] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, 1990.

[15] L. Moss. Parametric corecursion. *Theoretical Computer Science*, 260(1–2):139–163, 2001.

Inflationary and Deflationary Fixed Points

Erich Grädel
Aachen University
graedel@informatik.rwth-aachen.de

Fixed point logics extend a basic logical formalism (like first-order logic, conjunctive queries, or propositional modal logic) by a constructor for forming *fixed points of relational operators.* The most influential fixed point formalisms in computer science have been concerned with least (and greatest) fixed points of monotone operators.

- The modal μ-calculus L_μ is the extension of propositional modal logic by least and greatest fixed points. This logic has been extensively studied, having acquired importance for a number of reasons. In terms of expressive power, it subsumes a variety of modal and temporal logics used in verification, in particular LTL, CTL, CTL*, PDL and also many logics used in other areas of computer science. On the other hand, L_μ has a rich theory, and is well-behaved in model-theoretic and algorithmic terms.

- LFP, the extension of first-order logic by least fixed points is of crucial importance in finite model theory and descriptive complexity, in particular due to its tight connection to polynomial-time computability. It relates to first-order logic in much the same way as L_μ relates to propositional modal logic.

In finite model theory and, to a lesser extent, in database theory, a number of other fixed point operators have been extensively studied, including inflationary, partial, nondeterministic and alternating fixed points. All of these have in common that they allow the construction of fixed points of operators that are not necessarily monotone.

In this talk, we will focus on inflationary and deflationary fixed point inductions and compare them to least and greatest fixed points. We will also show a number of examples and scenarios in which inflationary and deflationary fixed points arise in a natural way. Recall that in least fixed point logic we can write formulae $[\mathbf{lfp} R\bar{x} \,.\, \varphi(R,\bar{x})](\bar{a})$ expressing that $\bar{a}$ is in the least set R satisfying $R = \{\bar{x} : \varphi(R,\bar{x})\}$. We can do this, provided that the relation variable R appears only positively in φ. This guarantees that, on every structure, the operator $F_\varphi : R \mapsto \{\bar{x} : \varphi(R,\bar{x})\}$ is monotone and therefore has a least (and a greatest) fixed point. Moreover, the fixed point can be obtained by an iterative process. Starting with the empty set, we repeatedly apply the operator F_φ and thus obtain an increasing (possibly transfinite) series of stages which converges to the desired least fixed point. A slightly different variant permits also simultaneous least fixed point inductions over several formulae, but it can be shown that this does not provide more expressive power.

Inflationary fixed points, on the other hand, can be built with formulae $\varphi(R,\bar{x})$ that need not be positive in R. Starting with the empty set, we can still define an increasing sequence of stages by iteratively taking the union of the current stage R with $F_\varphi(R)$. Again this sequence must eventually converge to a fixed point (not necessarily of F_φ, but of the operator $R \mapsto R \cup F_\varphi(R)$), which we call the inflationary fixed point of φ. In IFP we build formulae $[\mathbf{ifp}\ R\bar{x} \,.\, \varphi(R,\bar{x})](\bar{a})$ saying that $\bar{a}$ is contained in the inflationary fixed point of φ. The deflationary fixed point of $\varphi(R,\bar{x})$ is defined by a dual process, starting with $R = A^k$ and iteratively applying the operator $R \mapsto R \cap F_\varphi(R)$. This defines a decreasing sequence converging to a fixed point, called the deflationary fixed point of φ, which is also definable in IFP.

We review some of the known results on the logics LFP and IFP.

(1) Gurevich and Shelah have shown that on finite structures, LFP and IFP have the same expressive power. A recent result due to Kreutzer shows that this equivalence of LFP and IFP also extends to infinite structures.

(2) On ordered finite structures, LFP and IFP express precisely the properties that are decidable in polynomial time.

(3) Simultaneous least or inflationary inductions do not provide more expressive power than simple inductions.

(4) The complexity of evaluating a formula ψ in LFP or IFP on a given finite structure is polynomial in the size of the structure, but exponential

in the length of the formula. For formulae with a bounded number k of variables, the evaluation problem is PSPACE-complete, even for $k = 2$ and on fixed (and very small) structures. If, in addition to bounding the number of variables one also forbids parameters in fixed point formulae, the evaluation problem for LFP is computationally equivalent to the model checking problem for L_μ which is known to be in NP $\cap$ Co-NP, in fact in UP $\cap$ Co-UP, and hard for PTIME. It is an open problem whether this problem can be solved in polynomial time.

We also note that even though IFP does not provide more expressive power than LFP on finite structures, it is often more convenient to use inflationary inductions in explicit constructions. The advantage of using IFP is that one is not restricted to inductions over positive formulae. A non-trivial case in point is the formula defining an order on the k-variable types in a finite structure, an essential ingredient of the proof of the Abiteboul-Vianu Theorem, saying that least and partial fixed point logics coincide if and only if PTIME = PSPACE. Furthermore, IFP is more robust, in the sense that inflationary fixed points are well-defined, even when other, non-monotone, operators are added to the language.

Inflationary Inductions in Modal Logic. Given the close relationship between LFP and IFP on finite structures, and the importance of the μ-calculus, it is natural to study also the properties and expressive power of inflationary fixed points in modal logic. We define a modal iteration calculus, MIC, by extending basic multi-modal logic with simultaneous inflationary inductions. Given formulae $\varphi_1, \dots, \varphi_k$ we can build formulae **ifp** $X_i : [X_1 \leftarrow \varphi_1, \dots, X_k \leftarrow \varphi_k]$ that construct sets by a simultaneous inflationary induction. At each stage α, we have a tuple of sets $X_1^\alpha, \dots, X_k^\alpha$. Substituting these into the formulae $\varphi_1, \dots, \varphi_k$ we obtain a new tuple of sets, which we *add* to the existing sets $X_1^\alpha, \dots, X_k^\alpha$, to obtain the next stage.

It is clear that MIC is a modal logic in the sense that it is invariant under bisimulation. In fact, on every class of bounded cardinality, inflationary fixed points can be unwound to obtain equivalent infinitary modal formulae. As a consequence, MIC has the tree model property. It is also clear that MIC is at least as expressive as L_μ. The following natural questions now arise.

(1) Is MIC more expressive than L_μ?

(2) Does MIC have the finite model property?

(3) What are the algorithmic properties of MIC? Is the satisfiability problem decidable? Can model checking be performed efficiently (as efficiently as for L_μ)?

(4) Can we eliminate, as in the μ-calculus and as in IFP, simultaneous inductions without losing expressive power?

(5) What is the relationship of MIC with monadic second-order logic (MSO) and with finite automata? Or more generally, what are the 'right' automata for MIC?

(6) Is MIC the bisimulation-invariant fragment of any natural logic (as L_μ is the bisimulation-invariant fragment of MSO?)

We provide answers to most of these questions. Although IFP and LFP have equal expressive power, the situation for fixed point extensions of modal logic is quite different. The modal iteration calculus MIC has much greater expressive power than the μ-calculus. Greater expressive power comes at a cost: the calculus is algorithmically much less manageable. In particular, we establish the following results:

(1) There exist MIC-definable languages that are not regular. Hence MIC is more expressive than the μ-calculus, and does not translate to monadic second-order logic.

(2) MIC does not have the finite model property.

(3) The satisfiability problem for MIC is undecidable. In fact, it is not even in the arithmetic hierarchy.

(4) The model checking problem for MIC is PSPACE-complete.

(5) Simultaneous inflationary inductions do provide more expressive power than simple inflationary inductions. Nevertheless the algorithmic intractability results for MIC apply also to MIC without simultaneous inductions.

(6) There are bisimulation-invariant polynomial time properties that are not expressible in MIC.

(7) All languages in DTIME($O(n)$) are MIC-definable.

No doubt, these properties exclude MIC as a candidate logic for hardware verification. On the other hand, the present study is an investigation into the structure of the inflationary fixed point operator and may suggest tractable fragments of the logic MIC, which involve crucial use of an inflationary operator, just as logics like CTL and alternation-free L_μ carve out efficiently tractable fragments of L_μ. In any case, it delineates the differences between inflationary and least fixed point constructs in the context of modal logic.

(This is joint work with Anuj Dawar and Stephan Kreutzer)

Monadic Datalog on Trees

Martin Grohe

University of Edinburgh

Abstract

Semi-structured data, best-known in the syntax of XML, have caused a significant paradigm shift in the field of database systems, and have also been one of the central research topics in database theory over the last five years. While classical relational databases can be described as relational structures, XML-documents are best modelled by unranked trees. We study the problem of evaluating unary, or node-selecting, queries on trees. Node-selecting queries are not only of interest as basic queries in their own right, but are also an important building block for more complex queries. In particular, the node-selecting path query language XPath is at the core of several major XML-related technologies, such as XML Query, XML Schema, and XSLT, the principal query language, schema definition formalism, and stylesheet language for XML, respectively.

Even though undoubtedly very important in practice, from a theoretical perspective XPath seems to be a very ad-hoc language that leaves a lot to be desired. Monadic second-order logic (MSO) on trees, on the other hand, is well-known to have beautiful theoretical properties. In particular, it has well-balanced expressive power in that it is expressive enough for most purposes, but on the other hand still has good algorithmic properties due to its connection with tree automata. Indeed, MSO has been proposed as a "benchmark" for the expressive power of node-selecting XML query languages (Neven and Schwentick 2000). Nevertheless, MSO itself is not suitable as a practical query language because it allows to express very complex queries very concisely, which makes the query evaluation problem highly intractable. But there are nice languages which have the same expressive power as MSO on trees, but admit much more efficient query evaluation. The modal mu-calculus may be seen as an example of such a language (at least on ranked trees). In the context of querying XML, the most promising such language is monadic datalog. It has the same expressive power as MSO, but admits query evaluation in time linear in both the size of the datalog program and the size of the tree (Gottlob and Koch 2002).

In this talk, I will present recent results on the expressiveness and complexity of monadic datalog and related languages. Even though monadic datalog does have the same expressive power as MSO (on trees), there is no elementary translation from MSO into monadic datalog - we may say that MSO is non-elementarily more concise. We will look at such "conciseness" results more closely and place logics such as monadic least-fixed point logic and stratified monadic datalog into the picture. On the algorithmic side, we will show that the containment problem for monadic datalog on trees is in EXPTIME (and thus EXPTIME-complete). Furthermore, we

will discuss a new automata based algorithm for evaluating unary monadic datalog queries which has the nice property that it has to read the input tree only twice (in postorder), which is of great advantage when evaluating queries on documents that are too large to fit into main memory. Indeed, a recent implementation of this algorithm by Christoph Koch turns out to be highly efficient in practice.

This is joint work with Markus Frick, Christoph Koch, and Nicole Schweikardt.

Monadic fusion of functional programs

Claus Jürgensen[*][†]
Faculty of Computer Science
Dresden University of Technology
D-01062 Dresden, Germany

March 17, 2003

We present a new fusion technique to transform functional programs and prove its correctness. Instead of the catamorphism (*i.e.* the unique algebra morphism from an initial algebra) which is used in the 'acid rain theorem' we rather use the unique monad morphism from a free monad. Moreover we demonstrate how to use our fusion theorem to compose classes of tree transducers.

This paper is a shortened version of [Jür02] where more details and all the proofs can be found.

1 Introduction

This paper is about a program transformation of functional programs called *fusion*. Consider three algebraic types A, B, and C and two recursive programs f and g with typing $C \xleftarrow{f} B \xleftarrow{g} A$. We call a program $h : C \leftarrow A$ a **fusion** of the **consumer** f and the **producer** g if two conditions are satisfied:

(i) $[\![f]\!] \cdot [\![g]\!] = [\![h]\!]$, and

(ii) the intermediate data-structure B does not occur in h.

The condition (i) is the correctness of the fusion w.r.t. the denotational semantics $[\![.]\!]$. If the semantics is compositional it can be trivially satisfied by setting $h = f \cdot g = \lambda\, x \rightarrow f(g\, x)$. The essential point is condition (ii): the elimination of the intermediate data-structure.

Various fusion techniques are known, *e.g.*: *deforestation* [Wad90], *short cut fusion* [GLP93, TM95, Gil96, Joh01], or *syntactic composition of tree transducers (and attribute grammars)* [Eng75, Eng80, Fül81, CF82, EV85, Gie88, CDPR97b, CDPR97a, Küh98, FV98, KV01]. Each of these has its own advantages and disadvantages.

We combine the 'syntactic composition of tree transducers' [KV01] on the one hand side and 'short cut fusion' [GLP93] on the other hand side.

Short cut fusion is based on the 'cata/build-rule' or 'cata/augment-rule' [Joh01] or 'acid rain theorem' [TM95]. Therefore it is necessary to represent the recursive functions as catamorphisms. A catamorphism is a generalization of the well known list-function *foldr* for arbitrary

[*]Email: `Claus.Juergensen@Inf.TU-Dresden.DE`

[†]Supported by the postgraduate program 'Specification of discrete processes and systems of processes by operational models and logics' (GRK 334/2) of the German Research Community (DFG)

algebraic data types. In terms of category theory a catamorphism is the unique algebra morphism from an initial algebra.

We have invented a new fusion technique using monads: instead of a catamorphism we use the unique monad morphism from a free monad.

Consider the small Haskell program:

```
data Nat      = Zero | Succ Nat
data Bool     = False| True

even  Zero     = True
even (Succ n)  = odd  n
odd   Zero     = False
odd  (Succ n)  = even n
```

The latter four equations define the two mutually recursive functions *even* and *odd*. We can view this system of equations as a function[1]:

$$\begin{aligned} \varrho_X : \mathsf{T}_\Delta(\mathsf{Q}X) &\leftarrow \mathsf{Q}(\Sigma X), \\ \mathit{True} &\mapsfrom \mathit{even}\ \mathit{Zero}, \\ \mathit{odd}\ n &\mapsfrom \mathit{even}(\mathit{Succ}\ n), \\ \mathit{False} &\mapsfrom \mathit{odd}\ \mathit{Zero}, \\ \mathit{even}\ n &\mapsfrom \mathit{odd}(\mathit{Succ}\ n) \end{aligned}$$

where $X = \{n\}$ is the set of variables. The endofunctors Σ, Δ, and Q describe the application of ranked symbols from $\{\mathit{Zero}^{(0)}, \mathit{Succ}^{(1)}\}$, $\{\mathit{True}^{(0)}, \mathit{False}^{(0)}\}$, and $\{\mathit{even}^{(1)}, \mathit{odd}^{(1)}\}$, respectively, to a set (*e.g.* $\Sigma X = \{\mathit{Zero}\} \cup \{\mathit{Succ}\ x | x \in X\}$). The functor T_Δ constructs all Δ-trees over a set X: $\mathsf{T}_\Delta X = \biguplus_{k\in\mathbb{N}_0} \Delta^k X \cong X + \Delta(\mathsf{T}_\Delta X)$.

It is possible to show that the function ϱ_X is natural in X and thus we have a natural transformation:

$$\varrho : \mathsf{T}_\Delta \cdot \mathsf{Q} \dot{\leftarrow} \mathsf{Q} \cdot \Sigma$$

which we call the **rule** of the functional program. Using some category theory magic like adjoint functors we can equivalently transform this rule into the form:

$$\varrho' : |\mathsf{H}\mathbb{T}_\Delta| \dot{\leftarrow} \Sigma$$

where $\mathbb{T}_\Delta = (\mathsf{T}_\Delta, \eta, \mu)$ denotes the free monad over Δ, H is an endofunctor, and $|\bullet|$ the forgetful functor mapping a monad onto its underlying endofunctor. A rule in the latter form is the main ingredient of a so called **monadic transducer** which we introduce in Definition 5.1.1.

Using the universal property of a free monad we can define a denotational semantics for monadic transducers. Moreover, we have proved a new fusion theorem for monadic transducers (Theorem 5.2.2) similar to the 'acid rain theorem'.

Our construction depends on the syntactic structure of the functional programs f and g we want to compose. We use syntactic classes of *tree transducers* to describe the necessary syntactic form of the programs. A tree transducer [Rou68] is a finite tree automaton with in- and output. Its integral part is a set of rules. Some classes of tree transducers can be viewed as syntactic fragments of functional programming languages. Our example Haskell program is a top-down tree transducer which has the two states *even* and *odd*.

The composition of top-down tree transducers is an instance of short cut fusion [JV01]. But for more complicated tree transducers we have not been able to apply short cut fusion, and that is why we invented the monadic transducer.

[1]Please forgive us for drawing all arrows from right to left. In Subsection 2.1 we explain why we prefer it this way.

Moreover, we are interested in the question, whether syntactic classes of tree transducers are closed under fusion. This question has been answered (positively or negatively) for many classes of tree transducers. The constructions and proofs of the classical results differ depending on the specific class of tree transducers investigated. Using our monadic transducer we can describe many kinds of tree transducers in a uniform way. Once modeled as a monadic transducer, it is easy to do a fusion and then inspect whether the result is a tree transducer of a specific class.

We will show how to compose *homomorphism top-down, top-down,* and *macro tree transducers* with our new approach. In [Jür02] we show how it is possible to extend our new approach to the fusion of *high-level tree transducers, top-down tree-series transducers,* and *bottom-up tree transducers.* Even though we use some esoteric category theory, our results will be down-to-earth constructions which are applicable to transform real functional programs (see Figure 4).

2 Preliminaries

2.1 Functions and arrows

We denote the fact that a function f maps to a set A from a set B by $B = \operatorname{dom} f$ and $A = \operatorname{cod} f$ or by the relation $f : A \leftarrow B$. We will use this notation for a *morphism* f to an *object* A from an object B as well. A function is nothing else than a morphism in the category $\mathcal{S}et$. In order to avoid parentheses we will use the conventions $fx = f(x)$ and $\mathsf{F}fx = (\mathsf{F}f)x$ for function applications. The composition $f \cdot g : A \leftarrow C$ of two functions $f : A \leftarrow B$ and $g : B \leftarrow C$ is defined by $\forall x \in C.\, (f \cdot g)x = f(gx)$. This is the reason why all our arrows point to the left[2]:

$$\begin{tikzcd} A & B \arrow[l, "f"'] & C \arrow[l, "g"'] \arrow[ll, bend left, "f \cdot g"] \end{tikzcd}$$

We assume that function application binds stronger than function composition.

2.2 Category theory

We will use the following notions from category-theory: *(bi/endo)functor, natural transformation, horizontal/vertical composition, initial/final object, (co)product, projection, injection, exponent, (initial)* F*-algebra, universal arrow, (semi-)concrete category, free object, adjunction, monad, monadic,* and *varietor.*

If possible and appropriate we will use the following fonts: $A, B, C, \ldots$ for objects; $f, g, h, \ldots$ for morphisms; $\mathsf{F}, \mathsf{G}, \mathsf{H}, \ldots$ for functors; $\sigma, \tau, \varphi, \ldots$ for natural transformations; $\mathcal{C}, \mathcal{D}, \mathcal{E}, \ldots$ for categories; and $\mathbb{T}, \mathbb{T}', \ldots$ for monads.

We refer to objects and morphisms of some category $\mathcal{C}$ as $\mathcal{C}$-objects and $\mathcal{C}$-morphisms and denote the classes of all objects and all morphisms of $\mathcal{C}$ by $\mathrm{Ob}\,\mathcal{C}$ and $\mathrm{Mor}\,\mathcal{C}$. The subclass of all $\mathcal{C}$-morphisms to A from B is denoted by $\mathcal{C}(A, B)$.[3]

We denote the meta-category of all categories (with functors as morphisms) by $\mathcal{CAT}$ and the meta-category of all functors to $\mathcal{C}$ from $\mathcal{D}$ (with natural transformations as morphisms) by $\mathcal{C}^{\mathcal{D}}$ (called **functor category**). We will almost always omit the word *meta* since it will make no difference for what we are doing. For the endofunctor category we use the abbreviations $\mathcal{E}nd\,\mathcal{C} = \mathcal{C}^{\mathcal{C}}$ and $\mathcal{E}nd^2\,\mathcal{C} = \mathcal{E}nd(\mathcal{E}nd\,\mathcal{C})$.

[2]Arrows pointing to the right are consistent with the commuted composition $g\,;f = f \cdot g$.

[3]Notice, that this is more often denoted by $\mathcal{C}(B, A)$ or $\mathrm{Hom}_{\mathcal{C}}(B, A)$. Our notation is consistent with arrows pointing to the left which we use.

For every object A we denote the identity morphism by id_A or just id. The composition in a category is usually denoted by $f \cdot g$. The only exception will be the vertical composition of natural transformations denoted by $\sigma * \tau$ in order to distinguish it from the horizontal composition $\sigma \cdot \tau$.

For all $\mathcal{C} \xleftarrow{\mathsf{F},\mathsf{F}'} \mathcal{D} \xleftarrow{\mathsf{G},\mathsf{G}'} \mathcal{E}$ and all $\sigma : \mathsf{F} \dot{\leftarrow} \mathsf{F}'$ and $\tau : \mathsf{G} \dot{\leftarrow} \mathsf{G}'$ we write $(\sigma\mathsf{G})_X = \sigma_{\mathsf{G}X}$ and $(\mathsf{F}\tau)_X = \mathsf{F}(\tau_X)$. Then the **vertical composition** of the natural transformations σ and τ is given by $\sigma * \tau = \sigma\mathsf{G} \cdot \mathsf{F}'\tau = \mathsf{F}\tau \cdot \sigma\mathsf{G}'$. We denote the meta-category of all categories with all natural transformations as morphisms and composition $*$ by $\mathbb{CAT}$ where $\mathbb{CAT}(\mathcal{C},\mathcal{D}) = \mathrm{Mor}\,\mathcal{C}^{\mathcal{D}}$. The class of all natural transformations with horizontal and vertical composition is a 2-category.

We denote **coproducts** by $A+B$ or $\coprod_i A_i$, **products** by $A \times B$ or $\prod_i A_i$, and **exponents** by $A \Leftarrow B$ or A^B. If a category has finite coproducts we call it a **cocartesian** category.

The symbols for (co)products and exponents will also be used for the related functors (*e.g.* $\Leftarrow : \mathcal{C} \leftarrow \mathcal{C} \times \mathcal{C}^{\mathrm{op}}$) where we write the bifunctors $+$, $\times$, and $\Leftarrow$ as infix binary operators. We will denote the pointwise lifting of these bifunctors to the functor category by the same symbol (*e.g.* $(\mathsf{F} + \mathsf{G})f = \mathsf{F}f + \mathsf{G}f$). We use the following names for classes of functors defined from $+$, $\times$ and $\Leftarrow$ by gramars: **cocartesian** functors:

$$\mathsf{F}X ::= X \mid \mathsf{F}_1X + \mathsf{F}_2X,$$

bicartesian functors:

$$\mathsf{F}X ::= X \mid \mathsf{F}_1X + \mathsf{F}_2X \mid \mathsf{F}_1X \times \mathsf{F}_2X,$$

and **polynomial** functors: $\mathsf{F}X ::=$

$$A \mid X \mid \mathsf{F}_1X + \mathsf{F}_2X \mid \mathsf{F}_1X \times \mathsf{F}_2X \mid \mathsf{F}_1 \Leftarrow A.$$

For every category $\mathcal{C}$ we denote the identity functor by $\mathsf{Id}_{\mathcal{C}}$ or just Id.

For every object A we denote the constant functor which maps onto id_A by $\underline{A}$. Notice that the function $\underline{\ \cdot\ }$ is a functor $\underline{\ \cdot\ } : \boldsymbol{End}\,\mathcal{C} \leftarrow \mathcal{C}$ defined on $\mathcal{C}$-morphisms f an $\mathcal{C}$-objects X by $\underline{f}_X = f$.

We denote a **semi-concrete** category built upon $\mathcal{C}$ by $(\mathcal{D}, \mathsf{U})$ where $\mathsf{U} : \mathcal{C} \leftarrow \mathcal{D}$. If U is faithful we call $(\mathcal{D}, \mathsf{U})$ a **concrete category** and U its **forgetful functor.** A **semi-concrete** functor $\mathsf{F} : (\mathcal{D}, \mathsf{U}) \leftarrow (\mathcal{D}', \mathsf{U}')$ is a functor $\mathsf{F} : \mathcal{D} \leftarrow \mathcal{D}'$ such that $\mathsf{U} \cdot \mathsf{F} = \mathsf{U}'$ holds. If $(\mathcal{D}, \mathsf{U})$ is concrete then we call F a **concrete functor**. Notice, that concrete functors are uniquely determined by their values on objects.

We denote an **adjunction** by $(\eta, \varepsilon) : \mathsf{F} \dashv \mathsf{G} : \mathcal{C} \leftarrow \mathcal{D}$ or just by $\mathsf{F} \dashv \mathsf{G}$ where $\mathsf{F} : \mathcal{C} \leftarrow \mathcal{D}$, $\mathsf{G} : \mathcal{D} \leftarrow \mathcal{C}$, $\eta : \mathsf{G} \cdot \mathsf{F} \dot{\leftarrow} \mathsf{Id}$, and $\varepsilon : \mathsf{Id} \dot{\leftarrow} \mathsf{F} \cdot \mathsf{G}$.

If a concrete category $(\mathcal{D}, \mathsf{U})$ has free objects, *i.e.* the forgetful functor is right adjoint, then we call the left adjoint of U the **free-functor.**

3 Tree transducers

A tree transducer [Rou68] is a finite tree automaton with input and output. We consider tree transducers for two main reasons:

(i) We view tree transducers as a syntactic fragment of a functional programming language. Then we can use the nomenclature of the theory of tree transducers for functional programs.

(ii) It turns out that our monadic fusion theorem (Theorem 5.2.2) has a form which makes it possible to reason about the fusion of classes of functions. This is one of the problems investigated in the theory of tree transducers.

We will only consider deterministic total tree transducers, *i.e.* the rules of the tree transducers are functions.

3.1 The rule of a tree transducer

We have seen in the introduction how to describe the defining equations of a functional programs as one function which we called the *rule* of the program. The type of the rule describes the syntactic structure of the program. In the case that the functional program is a tree transducer, we have the following:

3.1.1 Proposition (tree transducer rules are natural transformations). Every tree transducer rule can be uniquely extended to a natural transformation, and vice versa every natural transformation of the appropriate type can be restricted to a tree transducer rule. ◇

And thus we can define tree transducers simply by giving the types of natural transformations.

Before we can do so we need one more definition in order to express *applicative terms* [Dam82], *i.e.* terms, where some subterms are treated as functions which can be applied to other terms.

3.1.2 Definition. Let $\mathcal{C}$ be a cartesian closed category and $I \in \mathrm{Ob}\,\mathcal{C}$. We define the functor A_I : $End\,\mathcal{C} \leftarrow \mathcal{C}$ by

$$\forall f, g \in \mathrm{Mor}\,\mathcal{C}.\ \mathsf{A}_I f g = (g \Leftarrow id_I) \times f.$$

Obviously A_I is polynomial.

For every pair of objects $X, Y \in \mathrm{Ob}\,\mathcal{C}$ we have $\mathsf{A}_I XY = Y^I \times X$. For $\mathcal{C} = \mathcal{S}et$ and $k \in \mathbb{N}_0$ we define $\mathsf{A}_k XY = \{x y_1 \cdots y_k \mid x \in X \wedge y_i \in Y\} \cong Y^k \times X \cong \mathsf{A}_{\{1,\ldots,k\}} XY$. ◇

We will not give a precise definition of a tree transducer. We will rather define some classes of tree transducers (and the respective classes of computable functions) by stating the type of their *rule* in Table 1 where Σ, Δ are polynomial $\mathcal{S}et$-endofunctors; Q is a cocartesian $\mathcal{S}et$-endofunctor; X *(recursion variables)*, Y *(context variables)* are sets; $k \in \mathbb{N}$; A is a complete semiring; and $\mathbb{B}$ is the boolean semiring. According to Proposition 3.1.1 the latter is an equivalent representation of a tree transducer.

syntactic class	class of computable functions	type of rules		
homomorphism	*HOM*	$\mathsf{T}_\Delta \quad X$	$\leftarrow$	ΣX
top-down	*TOP*	$\mathsf{T}_\Delta \quad (\mathsf{Q}X)$	$\leftarrow$	$\mathsf{Q}(\Sigma X)$
simple basic macro	*sb-MAC*	$\mathsf{T}_\Delta(\quad Y + \mathsf{A}_k(\mathsf{Q}X) \quad Y\ )$	$\leftarrow$	$\mathsf{A}_k(\mathsf{Q}(\Sigma X))Y$
basic macro	*b-MAC*	$\mathsf{T}_\Delta(\mathsf{T}_\Delta Y + \mathsf{A}_k(\mathsf{Q}X)(\mathsf{T}_\Delta Y))$	$\leftarrow$	$\mathsf{A}_k(\mathsf{Q}(\Sigma X))Y$
macro	*MAC*	$\mathsf{T}_{\mathsf{T}_\Delta + \mathsf{A}_k(\mathsf{Q}X)} Y$	$\leftarrow$	$\mathsf{A}_k(\mathsf{Q}(\Sigma X))Y$
top-down tree-series	TOP_A	$A\langle\langle \mathsf{T}_\Delta(\mathsf{Q}X)\rangle\rangle$	$\leftarrow$	$\mathsf{Q}(\Sigma X)$
nondeterministic top-down	$TOP_\mathbb{B}$	$\mathbb{B}\langle\langle \mathsf{T}_\Delta(\mathsf{Q}X)\rangle\rangle$	$\leftarrow$	$\mathsf{Q}(\Sigma X)$
bottom-up	*BOT*	$\mathsf{Q}(\mathsf{T}_\Delta X)$	$\leftarrow$	$\Sigma(\mathsf{Q}X)$

Table 1: Some classes of tree transducers

3.1.3 Example (macro tree transducer). Consider the Haskell program:

```
data Nat = Zero| Succ Nat
data [α]  = []   | α : [α]


reverse x  = let
                rev []        ys = ys
                rev (x : xs)ys = rev xs (x : ys)
             in rev x []
```

```
expo x  =  let
                exp (Succ x)y =  exp x (exp x y)
                exp Zero     y =  Succ y
            in exp x Zero

append x  =  let
                app (x : xs)ys =  x  :  (app xs ys)
                app []        ys =  ys
            in app x []
```

Then the program *reverse* is a non-simple basic macro tree transducer, the program *expo* is a simple non-basic macro tree transducer (This one is not in Table 1.), and the program *append* is a simple basic macro tree transducer. ◇

4 Monads and monad transformers

A **monad** $\mathbb{T} = (\mathsf{T}, \eta, \mu)$ on $\mathcal{C}$ is a triple consisting of an endofunctor T and two natural transformations $\eta : \mathsf{T} \dot{\leftarrow} \mathsf{Id}$ and $\mu : \mathsf{T} \dot{\leftarrow} \mathsf{T}^2$ such that $\mu \cdot \mathsf{T}\eta = id_{\mathsf{T}} = \mu \cdot \eta\mathsf{T}$ and $\mu \cdot \mathsf{T}\mu = \mu \cdot \mu\mathsf{T}$ holds. The intuition of a monad, that we will need, is that it can be viewed as a description of a recursive data structure together with a notion of substitution.

4.1 Tree monads and free monads

The easiest example for a monad is the **trivial monad** $\mathbb{ID}_{\mathcal{C}} = (\mathsf{Id}_{\mathcal{C}}, id_{\mathsf{Id}_{\mathcal{C}}}, id_{\mathsf{Id}_{\mathcal{C}}})$ on a category $\mathcal{C}$.

As an other example we give the monad of all trees over a ranked alphabet Σ. The endofunctor T is given by $\mathsf{T}X = \mathsf{T}_\Sigma X$, where $\mathsf{T}_\Sigma X$ denotes the set of Σ-trees over the set of variables X. Then $\eta_X : \mathsf{T}_\Sigma X \leftarrow X$ describes the embedding of variables into trees $X \subseteq \mathsf{T}_\Sigma X$ and $\mu_X : \mathsf{T}_\Sigma X \leftarrow \mathsf{T}_\Sigma(\mathsf{T}_\Sigma X)$ describes substitution in the following way: For an interpretation of variables $i : \mathsf{T}_\Sigma Y \leftarrow X$ the substitution operator is given by $\mu_Y \cdot \mathsf{T}_\Sigma i : \mathsf{T}_\Sigma Y \leftarrow \mathsf{T}_\Sigma X$. In the special case $\{x_1, \ldots, x_k\} \subseteq X = Y$ where i is defined using some $t_1, \ldots, t_k \in \mathsf{T}_\Sigma X$ by $ix_j = t_k$ (and $iy = y$ otherwise) we have $\mu_Y \cdot \mathsf{T}_\Sigma i = [t_1/x_1, \ldots, t_k/x_k]$.

An endofunctor $\Sigma : \mathcal{C} \leftarrow \mathcal{C}$ is called a **varietor** if the concrete category of Σ-algebras $(\mathcal{C}^\Sigma, |\,.\,|)$ has free objects. It is a well known fact that polynomial $\mathcal{S}et$-endofunctors are varietors and that the free monads over these endofunctors describes free term-algebras together with the common term-substitution.

4.1.1 Definition (monad morphism, free monad). Let $\mathbb{T} = (\mathsf{T}, \eta, \mu)$ and $\mathbb{T}' = (\mathsf{T}', \eta', \mu')$ be monads over $\mathcal{C}$.

(i) A natural transformation $h : \mathsf{T} \dot{\leftarrow} \mathsf{T}'$ such that $\eta = h \cdot \eta'$ and $\mu \cdot (h * h) = h \cdot \mu'$ is called a **monad morphism** to $\mathbb{T}$ from $\mathbb{T}'$ and we write $h : \mathbb{T} \leftarrow \mathbb{T}'$.

(ii) It is easy to see that (i) gives rise to a category which we will denote by $\mathcal{M}nd\,\mathcal{C}$. Moreover this is a concrete category $(\mathcal{M}nd\,\mathcal{C}, |\,.\,|)$ built upon $\mathcal{E}nd\,\mathcal{C}$ where the forgetful functor $|\,.\,|$ maps a monad $(\mathsf{T}, \eta, \mu) \in \mathrm{Ob}(\mathcal{M}nd\,\mathcal{C})$ onto the underlying endofunctor T.

(iii) Let $\Sigma : \mathcal{C} \leftarrow \mathcal{C}$ be an endofunctor. A free object over Σ in $(\mathcal{M}nd\,\mathcal{C}, |\,.\,|)$ is called a **free monad** over Σ. We denote a free monad over Σ by $\mathbb{T}_\Sigma$ and its underlying functor by T_Σ. ◇

It is well known that varietors have free monads:

4.1.2 Theorem ([AHS90] Theorem 20.56). If $\Sigma : \mathcal{C} \leftarrow \mathcal{C}$ is a varietor, then $(\mathcal{C}^\Sigma, |\,.\,|)$ is monadic over $\mathcal{C}$ and the associated monad is a free monad generated by Σ. ◇

4.2 Monad transformers

One part of the monadic transducer, which we define later in Definition 5.1.1, is an endofunctor on a category of all monads over some category. Such a functor is sometimes called a *monad transformer* [Mog90].

Many different definitions of a monad transformer exist in the literature. In [Hin00] a monad transformer $(\mathsf{H}, \pi, \omega)$ is an endofunctor H mapping monads onto monads together with two natural transformations $\pi : \mathsf{H} \dot{\leftarrow} \mathsf{Id}$ (called **promote** or **lift**) and $\omega : \mathsf{Id} \dot{\leftarrow} \mathsf{H}$ (called **observe**). We will need a natural transformation $\omega : |\bullet|_0 \dot{\leftarrow} |\bullet|_0 \cdot \mathsf{H}$ to observe the final result of the monadic computation (see Definition 4.4.3). However, this function ω will not be part of our definition of a monad transformer.

In [JV01] we have used an *algebra transformer* (*cf.* [Fok92]) to formulate a generalized version of the 'acid-rain-theorem'. The rôle of this algebra transformer will be taken by an endofunctor mapping monads onto monads in our new fusion Theorem 5.2.2.

4.2.1 Definition. A **pointed functor** (F, π) on a category $\mathcal{C}$ is a pair consisting of a $\mathcal{C}$-endofunctor F and a natural transformation $\pi : \mathsf{F} \dot{\leftarrow} \mathsf{Id}_{\mathcal{C}}$. A **monad transformer** (H, π) on $\mathcal{C}$ is a pointed functor on $\mathit{Mnd}\,\mathcal{C}$. ◇

In the following two subsections we will see how to construct monad transformers from adjunctions or coproducts of monads:

4.3 Monad transformers from adjunctions

4.3.1 Lemma (composition of an adjunction and a monad). Let $(\eta, \varepsilon) : \mathsf{Q} \dashv \mathsf{U} : \mathcal{C} \leftarrow \mathcal{D}$ be an adjunction and $\mathbb{T} = (\mathsf{T}, \bar{\eta}, \mu)$ be a monad on $\mathcal{D}$. Then $\mathbb{T}' = (\mathsf{U} \cdot \mathsf{T} \cdot \mathsf{Q}, \mathsf{U}\bar{\eta}\mathsf{Q} \cdot \eta, \mathsf{U}(\mu \cdot \mathsf{T}\varepsilon\mathsf{T})\mathsf{Q})$ is a monad on $\mathcal{C}$. ◇

We have just seen the function $\mathsf{U} \cdot \mathsf{T} \cdot \mathsf{Q} \leftarrow\!\shortmid \mathsf{T}$. For the following it will be useful to give it a name:

4.3.2 Definition. Let $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$ be categories. We define the binary operator $\leftarrow\!\circ$ for every $\alpha \in \mathrm{Mor}\,\mathcal{C}^{\mathcal{D}}$ and $\beta \in \mathrm{Mor}\,\mathcal{E}^{\mathcal{F}}$ by

$$\forall \mathsf{H} \in \mathrm{Ob}\,\mathcal{D}^{\mathcal{E}}.\ (\alpha \leftarrow\!\circ\, \beta)_{\mathsf{H}} = \alpha * id_{\mathsf{H}} * \beta.$$

It is easy to see that $\leftarrow\!\circ$ is a bifunctor

$$\bullet \leftarrow\!\circ \bullet : (\mathcal{C}^{\mathcal{F}})^{\mathcal{D}^{\mathcal{E}}} \leftarrow \mathcal{C}^{\mathcal{D}} \times \mathcal{E}^{\mathcal{F}}$$

where the value of $\leftarrow\!\circ$ applied to a pair of objects $\mathsf{F} \in \mathrm{Ob}\,\mathcal{C}^{\mathcal{D}}$ and $\mathsf{G} \in \mathrm{Ob}\,\mathcal{E}^{\mathcal{F}}$ is given by

$$\forall \varphi \in \mathrm{Mor}\,\mathcal{D}^{\mathcal{E}}.\ (\mathsf{F} \leftarrow\!\circ\, \mathsf{G})\varphi = \mathsf{F}\varphi\mathsf{G}$$

where $\mathsf{F} \leftarrow\!\circ\, \mathsf{G}$ is a functor

$$\mathsf{F} \leftarrow\!\circ\, \mathsf{G} : \mathcal{C}^{\mathcal{F}} \leftarrow \mathcal{D}^{\mathcal{E}}$$

given on objects $\mathsf{H} \in \mathrm{Ob}\,\mathcal{D}^{\mathcal{E}}$ by

$$(\mathsf{F} \leftarrow\!\circ\, \mathsf{G})\mathsf{H} = \mathsf{F} \cdot \mathsf{H} \cdot \mathsf{G}.$$

Notice, that the latter makes $\leftarrow\!\circ$ to a bifunctor

$$\bullet \leftarrow\!\circ \bullet : \mathit{CAT} \leftarrow \mathit{CAT} \times \mathit{CAT}^{\mathrm{op}}.$$

4.3.3 Lemma. The binary operator $\leftarrow\!\circ$ is a functor

$$\bullet \leftarrow\!\circ \bullet : \mathbb{CAT} \leftarrow \mathbb{CAT} \times \mathbb{CAT}^{\mathrm{op}}$$

which is given on objects by

$$\forall \mathcal{C}, \mathcal{D} \in \mathrm{Ob}\,\mathbb{CAT}.\, \mathcal{C} \leftarrow\!\circ \mathcal{D} = \mathcal{C}^{\mathcal{D}}. \qquad \diamond$$

4.3.4 Definition. Let $\mathsf{Q} : \mathcal{C} \leftarrow \mathcal{D}$ be a left adjoint endofunctor. We use the construction from Lemma 4.3.1 to define a functor $\overline{\mathsf{Q}} : \mathit{Mnd}\,\mathcal{D} \leftarrow \mathit{Mnd}\,\mathcal{C}$ by

$$\forall (\mathsf{T}, \bar{\eta}, \mu) \in \mathrm{Ob}(\mathit{Mnd}\,\mathcal{C}).\, \overline{\mathsf{Q}}(\mathsf{T}, \bar{\eta}, \mu) = (\mathsf{U} \cdot \mathsf{T} \cdot \mathsf{Q}, \mathsf{U}\bar{\eta}\mathsf{Q} \cdot \eta, \mathsf{U}(\mu \cdot \mathsf{T}\varepsilon\mathsf{T})\mathsf{Q}),$$

$$\forall h \in \mathrm{Mor}(\mathit{Mnd}\,\mathcal{C}).\, \overline{\mathsf{Q}}h = \mathsf{U}h\mathsf{Q}$$

where $(\eta, \varepsilon) : \mathsf{Q} \dashv \mathsf{U}$ is an adjunction. Moreover $\overline{\mathsf{Q}}$ is a concrete functor

$$\overline{\mathsf{Q}} : (\mathit{Mnd}\,\mathcal{D}, |\bullet|) \leftarrow (\mathit{Mnd}\,\mathcal{C}, (\mathsf{U} \leftarrow\!\circ \mathsf{Q}) \cdot |\bullet|). \qquad \diamond$$

4.3.5 Lemma. The function $\bar{\bullet}$ from Definition 4.3.4 is a functor

$$\bar{\bullet} : \mathit{CAT} \leftarrow \mathit{LeftAdj}^{\mathrm{op}}$$

where *LeftAdj* denotes the subcategory of *CAT* where the morphisms are all left adjoint functors. $\diamond$

4.3.6 Lemma. The bifunctor $\leftarrow\!\circ$ preserves adjunctions: Let Q, U, Q', and U' be functors. Then

$$\mathsf{Q} \dashv \mathsf{U} \;\wedge\; \mathsf{Q}' \dashv \mathsf{U}' \implies (\mathsf{Q} \leftarrow\!\circ \mathsf{U}') \dashv (\mathsf{U} \leftarrow\!\circ \mathsf{Q}'). \qquad \diamond$$

4.3.7 Proposition. Let $\mathcal{C}$ be a category. The bifunctor

$$\leftarrow\!\circ : \mathit{End}^2\,\mathcal{C} \leftarrow \mathit{End}\,\mathcal{C} \times \mathit{End}\,\mathcal{C}$$

can be extended to a concrete bifunctor:

$$\bullet \leftarrow\!\circ \bullet : (\mathit{Mnd}(\mathit{End}\,\mathcal{C}), |\bullet|) \leftarrow (\mathit{Mnd}\,\mathcal{C}, |\bullet|) \times (\mathit{Mnd}\,\mathcal{C}, |\bullet|)$$

Proof. We define $\leftarrow\!\circ$ on monads (T, η, μ) and $(\tilde{\mathsf{T}}, \tilde{\eta}, \tilde{\mu})$ on $\mathcal{C}$ by

$$(\mathsf{T}, \eta, \mu) \leftarrow\!\circ (\tilde{\mathsf{T}}, \tilde{\eta}, \tilde{\mu}) = (\mathsf{T} \leftarrow\!\circ \tilde{\mathsf{T}}, \eta \leftarrow\!\circ \tilde{\eta}, \mu \leftarrow\!\circ \tilde{\mu})$$

and have to verify that it maps monads onto monads and monad morphisms onto monad morphisms. ∎

4.3.8 Lemma. Let $(\varepsilon, \eta) : \mathsf{Q} \dashv \mathsf{U} : \mathcal{C} \leftarrow \mathcal{D}$ be an adjunction where Q is a cocartesian endofunctor. Then $\overline{\mathsf{Q}}$ is a monad transformer.

Proof. Since Q is cocartesian we have a product $(\mathsf{Q} \xleftarrow{\iota_q} \mathsf{Id})_{q \in Q}$ where Q is a finite set. We claim that $(\overline{\mathsf{Q}}, \pi)$ is a pointed functor where for every monad $\mathbb{T} = (\mathsf{T}, \bar{\eta}, \mu)$ we define $\pi_{\mathbb{T}} = \mathsf{U}[\mathsf{T}\iota_q]_{q \in Q} \cdot \eta\mathsf{T}$ where $m = [\mathsf{T}\iota_q]_{q \in Q}$ denotes the unique mediating morphism satisfying $\forall q \in Q.\, m \cdot \iota_q = \mathsf{T}\iota_q$. We have to verify that π is natural in $\mathbb{T}$ and that π is a monad morphism. ∎

4.4 Monad transformers from coproducts of monads

The coproduct of monads on a category $\mathcal{C}$ is just the usual coproduct in the category $\boldsymbol{Mnd}\,\mathcal{C}$.
Colimits of monads have been studied in [Kel80]. Coproducts of monads have been used in [LG02a, LG02b] to construct monad transformers.

4.4.1 Lemma (coproduct of free monads). Let $\mathcal{C}$ be a cocartesian category and Σ and Δ be $\mathcal{C}$-varietors. Then:

$$\mathbb{T}_\Sigma + \mathbb{T}_\Delta \cong \mathbb{T}_{\Sigma+\Delta}$$

Proof. The free-functor mapping a varietor onto its free monad is left adjoint and thus preserves coproducts. ■

4.4.2 Definition and Lemma. Let $\mathcal{C}$ be a cocartesian category with initial object 0 and A a $\mathcal{C}$-object. Let us denote the left and right injections of binary coproducts by $\grave{\imath}$ and $\acute{\imath}$, respectively.

(i) $A^+ = \big((A + \,\centerdot\,), \acute{\imath}, [\grave{\imath}, id]\big)$ is a monad on $\mathcal{C}$. (In particular $0^+ \cong \mathbb{ID}$.)

(ii) The monad A^+ is free over $\underline{A}$ (*i.e.* $A^+ \cong \mathbb{T}_{\underline{A}}$).

(iii) The function $(\,\centerdot\,)^+$ is a functor

$$(\,\centerdot\,)^+ : \boldsymbol{Mnd}\,\mathcal{C} \leftarrow \mathcal{C}$$

defined on $\mathcal{C}$-morphisms f by $f^+ = \underline{f} + id$. ◇

4.4.3 Definition. Let $\mathcal{C}$ be a category and $I \in \mathrm{Ob}\,\mathcal{C}$.

(i) We define the functor $\Lambda_I : \mathcal{C} \leftarrow \boldsymbol{End}\,\mathcal{C}$ by

$$\forall \mathrm{T} \in \mathrm{Ob}(\boldsymbol{End}\,\mathcal{C}).\, \Lambda_I \mathrm{T} = \mathrm{T} I \quad \text{and}$$
$$\forall h \in \mathrm{Mor}(\boldsymbol{End}\,\mathcal{C}).\, \Lambda_I h = h_I.$$

(ii) We define the functor $|\,\centerdot\,|_I : \mathcal{C} \leftarrow \boldsymbol{Mnd}\,\mathcal{C}$ by $\Lambda_I \cdot |\,\centerdot\,|$ where $\big(\boldsymbol{Mnd}\,\mathcal{C}, |\,\centerdot\,|\big)$ is the concrete category of monads on $\mathcal{C}$ and $|\,\centerdot\,| : \boldsymbol{End}\,\mathcal{C} \leftarrow \boldsymbol{Mnd}\,\mathcal{C}$ the default forgetful functor mapping a monad onto its underlying endofunctor. ◇

4.4.4 Corollary. Let $\mathcal{C}$ be a category such that $\boldsymbol{Mnd}\,\mathcal{C}$ is cocartesian. Then:

(i)

$$(\,\centerdot\,)^+ : \big(\boldsymbol{Mnd}(\boldsymbol{Mnd}\,\mathcal{C}), |\,\centerdot\,| \cdot |\,\centerdot\,|_{\mathbb{ID}}\big) \leftarrow \big(\boldsymbol{Mnd}\,\mathcal{C}, |\,\centerdot\,|\big)$$

is a semi-concrete functor.

(ii) For every monad $\mathbb{T}$ on $\mathcal{C}$ the functor $(\mathbb{T} + \,\centerdot\,) = |\mathbb{T}^+|$ is a monad transformer: $\big((\mathbb{T} + \,\centerdot\,), \acute{\imath}\big)$ where $\acute{\imath}$ denotes a right injections into the coproduct of two monads.

(iii) For every $\mathcal{C}$-object A we have a concrete functor

$$(A^+ + \,\centerdot\,) : (\boldsymbol{Mnd}\,\mathcal{C}, |\,\centerdot\,|) \leftarrow \Big(\boldsymbol{Mnd}\,\mathcal{C}, (\mathsf{Id} \leftarrow\!\circ\, (A + \,\centerdot\,)) \cdot |\,\centerdot\,|\Big).$$ ◇

5 Monadic transducers

A monadic transducer is a generalization of a tree transducer described in terms of category theory. The advantage of monadic transducers is a higher level of abstraction which leads to much more elegant proofs and enables us to treat different kinds of tree transducers (homomorphism, top-down, high-level, tree-series, ...) in a unified framework. Monadic transducers can be used to give denotational semantics to fragments of functional programs. We will use this denotational semantics to prove the correctness of our monadic fusion.

5.1 Syntax and semantics of monadic transducers

5.1.1 Definition (monadic transducer). Let $\mathcal{C}$ be a category which has an initial object 0. $\mathbb{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho)$ is called a **monadic transducer** on $\mathcal{C}$ if

(i) $\mathsf{H} : \mathit{Mnd}\,\mathcal{C} \leftarrow \mathit{Mnd}\,\mathcal{C}$ (called **pattern**) is an endofunctor,

(ii) $\Sigma, \Delta : \mathcal{C} \leftarrow \mathcal{C}$ are varietors,

(iii) $\omega : |\bullet|_0 \stackrel{.}{\leftarrow} |\bullet|_0 \cdot \mathsf{H}$ (called **observe**), and

(iv) $\varrho : |\mathsf{H}\mathbb{T}_\Delta| \stackrel{.}{\leftarrow} \Sigma$ (called **rule**) are natural transformations. ◇

The intuition behind this definition is as follows: A *monadic transducer* is a particular functional program (or tree transducer) where: The two *varietors* describe the input and output data type, respectively. The *pattern* describes the syntactic structure or recursion pattern. The *rule* defines the program by a set of equations. And finally, the *observe* picks the desired value from the result of a mutual recursion.

We define the semantics of a monadic transducer in two phases:

5.1.2 Definition (generalized semantics of a monadic transducer). Let $\mathbb{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho)$ be a monadic transducer over $\mathcal{C}$. Since $\mathbb{T}_\Sigma$ is free over Σ, there exists a universal arrow $u_\Sigma : \mathsf{T}_\Sigma \stackrel{.}{\leftarrow} \Sigma$. Then there exists a unique monad morphism $(\!|\mathbb{M}|\!) : \mathsf{H}\mathbb{T}_\Delta \leftarrow \mathbb{T}_\Sigma$ such that $(\!|\mathbb{M}|\!) \cdot u_\Sigma = \varrho$ holds:

$$\begin{tikzcd}
 & \Sigma \arrow[ld, bend right, "\varrho"'] \arrow[d, dashed, "u^\Sigma"] \\
|\mathsf{H}\mathbb{T}_\Delta| & \mathsf{T}_\Sigma \arrow[l, "|(\!|\mathbb{M}|\!)|"'] \\
\mathsf{H}\mathbb{T}_\Delta & \mathbb{T}_\Sigma \arrow[l, "(\!|\mathbb{M}|\!)"']
\end{tikzcd}$$

The underlying natural transformation

$$(\!|\mathbb{M}|\!) : |\mathsf{H}\mathbb{T}_\Delta| \stackrel{.}{\leftarrow} \mathsf{T}_\Sigma$$

is called the **generalized semantics** of $\mathbb{M}$. The generalized semantics is independent from ω. However, it depends on the choice of the universal arrow u_Σ. To make things simpler, we choose for every varietor Σ a universal arrow u_Σ from Σ (to the free monad over Σ) and use this choice implicitly for the generalized semantics of every monadic transducer. To simplify our notation we will sometimes omit the forgetful functor $|\bullet|$ on morphisms. ◇

It is worth mentioning that 'being a monad morphism' is a natural property for the generalized semantics $(\!|\mathbb{M}|\!)$:

(i) $\eta = (\!|\mathbb{M}|\!) \cdot \eta'$ means that variables will be throughput and

(ii) $\mu \cdot ((\!|\mathbb{M}|\!) * (\!|\mathbb{M}|\!)) = (\!|\mathbb{M}|\!) \cdot \mu'$ states that $(\!|\mathbb{M}|\!)$ is compositional (or syntax directed).

5.1.3 Definition (semantics of a monadic transducer). Let $\mathbb{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho)$ be a monadic transducer over $\mathcal{C}$ which has an initial object 0. The **semantics** $[\![\mathbb{M}]\!] : \mathsf{T}_\Delta 0 \leftarrow \mathsf{T}_\Sigma 0$ of $\mathbb{M}$ is defined by

$$[\![\mathbb{M}]\!] = \omega_{\mathbf{T}_\Delta} \cdot |(\!|\mathbb{M}|\!)|_0 . \qquad ◇$$

5.2 Fusion of monadic transducers

5.2.1 Definition (fusion of monadic transducers). Let $\mathbb{M} = (\mathsf{H}, \Delta, \Gamma, \omega, \varrho)$ and $\mathbb{M} = (\mathsf{H}', \Sigma, \Delta, \omega', \varrho')$ be monadic transducers over $\mathcal{C}$ which has an initial object 0. The **fusion** $\mathbb{M} \cdot \mathbb{M}'$ of $\mathbb{M}$ and $\mathbb{M}'$ is the monadic transducer over $\mathcal{C}$ defined by

$$\mathbb{M} \cdot \mathbb{M}' = (\mathsf{H}' \cdot \mathsf{H}, \Sigma, \Gamma, \omega' * \omega, \mathsf{H}'\llparenthesis \mathbb{M} \rrparenthesis \cdot \varrho')$$

where $*$ denotes the vertical composition operator of natural transformations. Moreover we define for every $\mathcal{C}$-varietor Σ the **identity monadic transducer** by

$$\mathbb{ID}_\Sigma = (\mathsf{Id}, \Sigma, \Sigma, id, u_\Sigma). \qquad \diamond$$

5.2.2 Theorem. Let $\mathbb{M} = (\mathsf{H}, \Delta, \Gamma, \omega, \varrho)$ and $\mathbb{M}' = (\mathsf{H}', \Sigma, \Delta, \omega', \varrho')$ be monadic transducers over $\mathcal{C}$ which has an initial object 0. Then the following holds:

(i) $\llparenthesis \mathbb{ID} \rrparenthesis = id$, and

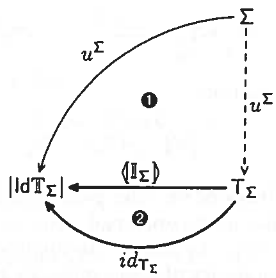
(ii) $\llparenthesis \mathbb{M} \cdot \mathbb{M}' \rrparenthesis = \mathsf{H}'\llparenthesis \mathbb{M} \rrparenthesis \cdot \llparenthesis \mathbb{M}' \rrparenthesis$.

(iii) The monadic transducers over $\mathcal{C}$ are the morphisms of a category $\mathcal{MT\,C}$ where composition is fusion and the objects are the class of all $\mathcal{C}$-varietors.

(iv) The semantics $[\![\,.\,]\!]$ is a functor

$$[\![\,.\,]\!] : \mathcal{C} \leftarrow \mathcal{MT\,C}.$$

Proof. (i) Consider the following diagram:



$$\begin{tikzcd}
& & \Sigma \arrow[dll, bend right, "u^\Sigma"'] \arrow[d, dashed, "u^\Sigma"] \\
{|\mathsf{Id}\mathbb{T}_\Sigma|} & ❶ & \mathsf{T}_\Sigma \arrow[ll, "\llparenthesis \mathbb{I}_\Sigma \rrparenthesis"'] \arrow[ll, bend left=50, "id_{\mathsf{T}_\Sigma}", "❷"']
\end{tikzcd}$$

The outside triangle around ❶❷ commutes trivially and ❶ commutes by definition of $\llparenthesis\,.\,\rrparenthesis$ (Definition 5.1.2). Thus ❷ also commutes, because u_Σ is universal.

(ii) Let $\tilde{\varrho} = \mathsf{H}'\llparenthesis \mathbb{M} \rrparenthesis \cdot \varrho'$. Consider the diagram in Figure 1.

(iii) We define the category $\mathcal{MT\,C}$ by

$$\mathsf{Ob}(\mathcal{MT\,C}) = \{\Sigma \mid \Sigma : \mathcal{C} \leftarrow \mathcal{C} \text{ is a varietor}\}$$
$$\mathcal{MT\,C}(\Delta, \Sigma) = \{\mathbb{M} \mid \mathbb{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho) \text{ is a monadic transducer}\}$$

where the identity for every $\Sigma \in \mathsf{Ob}(\mathcal{MT\,C})$ is the monadic transducer $\mathbb{ID}_\Sigma$ and composition is fusion. That the identities are neutral elements w.r.t. fusion is obvious. It remains to show that fusion is associative: Let $\mathbb{M}'' = (\mathsf{H}'', \Gamma, \Theta, \omega'', \varrho'')$, $\mathbb{M}' = (\mathsf{H}', \Delta, \Gamma, \omega', \varrho')$, and

$$\begin{tikzcd}
 & & & \Sigma \arrow[dlll, bend right=40, "\tilde{\varrho}"'] \arrow[dl, bend right, "\varrho'"'] \arrow[d, dashed, "u^{\Sigma}"] \\
{|(\mathsf{H}'\cdot\mathsf{H})\mathbb{T}_{\Gamma}|} & \text{❷} & {|\mathsf{H}'\mathbb{T}_{\Delta}|} \arrow[ll, "\mathsf{H}'(\!|\mathbb{M}|\!)"'] & \mathrm{T}_{\Sigma} \arrow[l, "(\!|\mathbb{M}'|\!)"'] \arrow[lll, bend left=40, "(\!|\mathbb{M}\cdot\mathbb{M}'|\!)", "\text{❸}"']
\end{tikzcd}$$

The outside triangle around ❶❷❸ and the triangle ❶ commute by definition (Definition 5.1.2). Obviously ❷ commutes by definition of $\tilde{\varrho}$. Thus ❸ also commutes, because u_Σ is universal.

Figure 1: Fusion of monadic transducers (generalized semantics)

$\mathbb{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho)$ be monadic transducers over $\mathcal{C}$. Then:

$$\begin{aligned}
&= (\mathbb{M}\cdot\mathbb{M}')\cdot\mathbb{M}'' \\
&= \big(\mathsf{H}''\cdot(\mathsf{H}'\cdot\mathsf{H}),\ \Sigma,\ \Theta,\ \omega''*(\omega'*\omega),\ \mathsf{H}''(\!|\mathbb{M}\cdot\mathbb{M}'|\!)\cdot\varrho''\big) \\
&= \big(\mathsf{H}''\cdot(\mathsf{H}'\cdot\mathsf{H}),\ \Sigma,\ \Theta,\ \omega''*(\omega'*\omega),\ \mathsf{H}''(\mathsf{H}'(\!|\mathbb{M}|\!)\cdot(\!|\mathbb{M}'|\!))\cdot\varrho''\big) \\
&= \big((\mathsf{H}''\cdot\mathsf{H}')\cdot\mathsf{H},\ \Sigma,\ \Theta,\ (\omega''*\omega')*\omega,\ (\mathsf{H}''\cdot\mathsf{H}')(\!|\mathbb{M}|\!)\cdot\mathsf{H}''(\!|\mathbb{M}'|\!)\cdot\varrho''\big) \\
&= \mathbb{M}\cdot\big(\mathsf{H}''\cdot\mathsf{H}',\ \Sigma,\ \Gamma,\ \omega''*\omega',\ \mathsf{H}''(\!|\mathbb{M}'|\!)\cdot\varrho''\big) \\
&= \mathbb{M}\cdot(\mathbb{M}'\cdot\mathbb{M}'')
\end{aligned}$$

(iv) With Definition 5.1.3 and (i) we calculate $[\![\mathbb{ID}_\Sigma]\!] = id_{\mathrm{T}_\Sigma}\cdot|(\!|\mathbb{ID}_\Sigma|\!)|_0 = id_{\mathrm{T}_\Sigma 0}$. Consider the diagram in Figure 2. Altogether we have that $[\![\,.\,]\!] : \mathcal{C} \leftarrow \mathcal{MTC}$ is a functor. ■

5.3 Monadic transducer homomorphisms

In order to interpret the results of fusions in Subsection 7.2 we will have to compare monadic transducers. We have two obvious notions of equivalence: Monadic transducers $\mathbb{M}$ and $\mathbb{M}'$ may be syntacticly equivalent ($\mathbb{M} = \mathbb{M}'$) or they may be semanticly equivalent ($[\![\mathbb{M}]\!] = [\![\mathbb{M}]\!]$). Artlessly, the former implies the latter.

It will become apparent (in Theorem 7.2.7) that a more subtle relation between monadic transducers is of use:

5.3.1 Definition. Let $\mathbb{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho)$ and $\mathbb{M}' = (\mathsf{H}', \Sigma, \Delta, \omega', \varrho')$ be monadic transducers. A natural transformation $\tau : \mathsf{H} \dot{\leftarrow} \mathsf{H}'$ such that

$$\omega_{\mathbf{T}_\Delta}\cdot|\tau_{\mathbf{T}_\Delta}|_0 = \omega'_{\mathbf{T}_\Delta} \quad\text{and}\quad \varrho = \tau_{\mathbf{T}_\Delta}\cdot\varrho'$$

holds is called a **monadic transducer homomorphism** to $\mathbb{M}$ from $\mathbb{M}'$ and we write it

$$\tau : \mathbb{M} \leftarrow \mathbb{M}'. \qquad \diamond$$

Obviously, 'being homomorphic' is a preorder on the class of monadic transducers, moreover it implies semantic equivalence as demonstrated in the following:

$$\begin{tikzcd}
{|(\mathsf{H}'\cdot\mathsf{H})\mathbb{T}_\Gamma|0} \arrow[d, "\omega'_{\mathsf{H}\mathbb{T}_\Gamma}"] \arrow[dd, bend right=60, "(\omega' * \omega)_{\mathbb{T}_\Gamma}"'] \arrow[dr, phantom, "❸"] & {|\mathsf{H}'\mathbb{T}_\Delta|0} \arrow[l, "|⦇\mathsf{M}⦈|_0"'] \arrow[d, "\omega'_{\mathbb{T}_\Delta}"] & \mathbb{T}_\Sigma 0 \arrow[l, "|⦇\mathsf{M}'⦈|_0"'] \arrow[ll, bend right=40, "|⦇\mathsf{M}\cdot\mathsf{M}'⦈|_0"', "❹"] \arrow[dl, dotted, "[\![\mathsf{M}']\!]", "❶"'] \arrow[ddll, dotted, bend left=30, "[\![\mathsf{M}\cdot\mathsf{M}']\!]", "❻"'] \\
{|\mathsf{H}\mathbb{T}_\Gamma|0} \arrow[d, "\omega_{\mathbb{T}_\Gamma}"] \arrow[l, phantom, "❺"] & \mathbb{T}_\Delta 0 \arrow[l, "|⦇\mathsf{M}⦈|_0"'] \arrow[dl, dotted, "[\![\mathsf{M}]\!]", "❷"'] & \\
\mathbb{T}_\Gamma 0 & &
\end{tikzcd}$$

The triangles ❶ and ❷ commute by definition and the square ❸ because $\omega' : |\cdot|_0 \leftarrow |\cdot|_0 \cdot \mathsf{H}'$ is natural. The triangle ❹ is just an instance of (ii)❸. The triangle ❺ and the outside triangle around ❶-❻ commute by definition. Thus ❻ also commutes.

Figure 2: Fusion of monadic transducers (semantics)

5.3.2 Theorem (monadic transducer homomorphisms preserve semantics). Let $\mathsf{M} = (\mathsf{H}, \Sigma, \Delta, \omega, \varrho)$ and $\mathsf{M}' = (\mathsf{H}', \Sigma, \Delta, \omega', \varrho')$ be monadic transducers. If there exists a monadic transducer homomorphism $\tau : \mathsf{M} \leftarrow \mathsf{M}'$ then $[\![\mathsf{M}]\!] = [\![\mathsf{M}']\!]$.

Proof. Consider the diagrams in Figure 3. Finally we calculate: $[\![\mathsf{M}]\!] = \omega_{\mathbb{T}_\Delta} \cdot ⦇\mathsf{M}⦈_0 = \omega'_{\mathbb{T}_\Delta} \cdot ⦇\mathsf{M}'⦈_0 = [\![\mathsf{M}']\!]$. ∎

6 Tree transducers as monadic transducers

We have already seen, that a tree transducer can be described by its *rule*. But for the different classes of tree transducers the types of these differ. A monadic transducer provides a uniform description for all classes of tree transducers.

6.1 Homomorphism tree transducers as monadic transducers

We start with the easiest case: the homomorphism tree transducer. The rule of a homomorphism tree transducer has the form $\mathsf{T}_\Delta X \xleftarrow{\varrho X} \Sigma X$ which can be abstracted from X (using Proposition 3.1.1) yielding the natural transformation: $\mathsf{T}_\Delta \xleftarrow{\varrho} \Sigma$. This is already the desired rule of the monadic transducer where the *pattern* and the *observe* are both trivial, *i.e.* identity functions.

6.1.1 Proposition. The homomorphism tree transducers are precisely the monadic transducers

$$\mathsf{M} = (\mathsf{Id}, \Sigma, \Delta, id, \varrho)$$

$$\begin{tikzcd}
 & {|H'\mathbb{T}_\Delta|0} \arrow[ld, "\omega'_{\mathbb{T}_\Delta}"'] \arrow[dd, "|\tau_{\mathbb{T}_\Delta}|_0"] & {|H'\mathbb{T}_\Delta|} \arrow[dd, "\tau_{\mathbb{T}_\Delta}"'] & & \text{❶} \\
T_\Delta 0 & \text{❹} & \text{❸} & T_\Sigma \arrow[lu, "(\!|M'|\!)"'] \arrow[ld, "(\!|M|\!)"] & & \Sigma \arrow[ll, dashed, "u^\Sigma"'] \arrow[llluu, bend right, "\varrho'"'] \arrow[llldd, bend left, "\varrho"] \\
 & {|H\mathbb{T}_\Delta|0} \arrow[lu, "\omega_{\mathbb{T}_\Delta}"] & {|H\mathbb{T}_\Delta|} & & \text{❷}
\end{tikzcd}$$

The triangles ❶ and ❷ commute by definition and the triangle around ❶❷❸ commutes according to the precondition. Thus the square around ❷❸ also commutes and since u_Σ is universal ❸ commutes. The triangle ❹ commutes according to the precondition.

Figure 3: Monadic transducer homomorphisms preserve semantics

over $\mathcal{S}et$ where Σ and Δ are polynomial. $\diamond$

6.2 Top-down tree transducers as monadic transducers

The rule of a top-down tree transducer has the form $T_\Delta(QX) \xleftarrow{\varrho X} Q(\Sigma X)$ where Q is cocartesian. This rule can be understood as a definition for a couple of functions (*e.g.* *even* and *odd* from the introduction). Alternatively we could define just *one* function whose results are tuples (*e.g.* $f\ x\ =\ (even\ x,\ odd\ x)$). Let us describe the tupling by U, *i.e.* UA is the set of all tuples with elements in A. Than the new rule would have the form

$$\begin{aligned} \mathsf{U}(T_\Delta(QX)) &\leftarrow \Sigma X, \\ (\mathit{True}, \mathit{False}) &\leftarrow\!\shortmid \mathit{Zero}, \\ (\mathit{odd}\ n, \mathit{even}\ n) &\leftarrow\!\shortmid \mathit{Succ}\ n. \end{aligned}$$

Let us formalize what we have done so far:

6.2.1 Lemma. Let $\mathcal{C}$ be a bicartesian category and $Q : \mathcal{C} \leftarrow \mathcal{C}$ a cocartesian functor. Then Q is left adjoint. $\diamond$

6.2.2 Lemma (flip rule type). Let $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{E}$ be categories, $T : \mathcal{C} \leftarrow \mathcal{E}$, $\Sigma : \mathcal{D} \leftarrow \mathcal{E}$, and $Q \dashv U : \mathcal{C} \leftarrow \mathcal{D}$. Then

$$\mathcal{C}^{\mathcal{E}}(T, Q \cdot \Sigma) \cong \mathcal{D}^{\mathcal{E}}(U \cdot T, \Sigma). \qquad \diamond$$

We start again with the rule $T_\Delta(QX) \xleftarrow{\varrho X} Q(\Sigma X)$. Abstraction from X (according to Proposition 3.1.1) yields the natural transformation: $T_\Delta \cdot Q \xleftarrow{\varrho} Q \cdot \Sigma$. The Lemmas 6.2.1 and 6.2.2 tell us that we can equivalently describe this rule by a natural transformation $U \cdot T_\Delta \cdot Q \xleftarrow{\varrho'} \Sigma$ where $Q \dashv U$. The latter is equal to $\overline{Q}\mathbb{T}_\Delta \xleftarrow{\varrho'} \Sigma$ by Definition 4.3.4. This is already the desired rule of the monadic transducer with *pattern* $\overline{Q}$.

Finally we have to decide which value the monadic transducer should output. In our example this means whether we want to compute the function *even* or *odd*. We do this with a projection

$\pi : \mathsf{Id} \leftarrow \mathsf{U}$ witch picks the desired value. Using $\mathsf{Q}\emptyset = \emptyset$ we can define the *observe* of the monadic transducer by $\omega = \pi| \centerdot |_\emptyset$.

Altogether we get the following:

6.2.3 Proposition. The top-down tree transducers are precisely the monadic transducers

$$\mathbb{M} = (\overline{\mathsf{Q}}, \Sigma, \Delta, \omega, \varrho)$$

over *Set* where Q is a cocartesian *Set*-endofunctor, Σ and Δ are polynomial, and $\omega = \pi| \centerdot |_\emptyset$ where π is a projection. $\diamond$

We have no room in this paper to describe the following in detail. The proofs and the precise constructions can be found in [Jür02]. In particular we will just ignore the *observe* of the monadic transducers.

The rule of a simple basic macro tree transducer has the form $\mathsf{T}_\Delta(Y + \mathsf{A}_I(\mathsf{Q}X)Y) \xleftarrow{(\varrho X)_Y} \mathsf{A}_I(\mathsf{Q}(\Sigma X))Y$ where A_I is the functor from Definition 3.1.2. As before in the top-down tree transducer case, we use adjunctions to move the functors Q and A_I. But this time the pattern is more complicated: We describe this by the functor $\leftarrow\!\!\circ$ from Definition 4.3.2 and $(\centerdot)^+$ from Definition 4.4.2 using Corollary 4.4.4.

6.2.4 Lemma. Let $\mathcal{C}$ be a category which has function spaces. Then

$$\mathsf{A}_I \dashv \Lambda_I : \boldsymbol{End}\,\mathcal{C} \leftarrow \mathcal{C}$$

where $\Lambda_I : \mathcal{C} \leftarrow \boldsymbol{End}\,\mathcal{C}$ is the functor from Definition 4.4.3. $\diamond$

6.2.5 Note. Let us illustrate the above theorem in the category *Set*: We use $\mathsf{A}_k XY = \{x\, y_1 \cdots y_k \mid x \in X \wedge y_i \in Y\} \cong Y^I \times X$ and $\Lambda_k \mathsf{T} = \{\lambda 1 \cdots k.t \mid t \in \mathsf{T}\{1,\ldots,k\}\} \cong \mathsf{T}\{1,\ldots,k\}$. Then we have an adjunction:

$$(\eta, \varepsilon) : \mathsf{A}_I \dashv \Lambda_I : \boldsymbol{End}\,\boldsymbol{Set} \leftarrow \boldsymbol{Set}$$

where $\eta_X x = (id_I, x)$ and $(\varepsilon_\mathsf{T})_Y(f,t) = \mathsf{T} f t$. Moreover, the function η and ε describe η-conversion:

$$\Lambda_k(\mathsf{A}_k X) = \{\lambda 1 \cdots k.t \mid t \in \mathsf{A}_k X\{1,\ldots,k\}\}$$

$$\begin{array}{rccll} \eta : & \Lambda_k \cdot \mathsf{A}_k & \leftarrow & \mathsf{Id} & \\ \eta_X : & \lambda 1 \cdots k.x\,1 \cdots k & \leftarrow\!\shortmid & x & (\eta\text{-conversion}) \end{array}$$

and β-reduction:

$$\mathsf{A}_k(\Lambda_k \mathsf{T})Y = \{(\lambda 1 \cdots k.t)\, y_1 \cdots y_k \mid t \in \mathsf{T}\{1,\ldots,k\} \wedge y_i \in Y\}$$

$$\begin{array}{rccll} \varepsilon : & \mathsf{Id} & \leftarrow & \Lambda_k \cdot \mathsf{A}_k & \\ (\varepsilon_\mathsf{T})_Y : & [y_i/i]_{i=1}^k t & \leftarrow\!\shortmid & (\lambda 1 \cdots k.t)\, y_1 \cdots y_k. & (\beta\text{-reduction}) \end{array}$$

$\diamond$

6.2.6 Proposition. The simple basic macro tree transducers are precisely the monadic transducers

$$\mathbb{M} = (\overline{\mathsf{A}_I \cdot \mathsf{Q}} \cdot (\mathsf{Id}^+ + \centerdot) \cdot (\centerdot \leftarrow\!\!\circ \mathbb{ID}), \Sigma, \Delta, \omega, \varrho)$$

over *Set* where I is a finite set, Q is a cocartesian *Set*-endofunctor, and Σ and Δ are polynomial. $\diamond$

The basic macro tree transducer case is just a little more complicated than the simple basic macro tree transducer: The only difference to Proposition 6.2.6 is that we have to replace the functor $(\centerdot \leftarrow\!\!\circ \mathbb{ID})$ by the functor $(\leftarrow\!\!\circ)$ defined by $\forall h.\ (\leftarrow\!\!\circ)h = h \leftarrow\!\!\circ h$.

6.2.7 Proposition. The basic macro tree transducers are precisely the monadic transducers

$$\mathbb{M} = \left(\overline{\mathsf{A}_I \cdot \mathsf{Q}} \cdot (\mathsf{Id}^+ + \,.\,) \cdot \langle \leftarrow\!\circ \rangle, \Sigma, \Delta, \omega, \varrho\right)$$

over *𝒮et* where I is a finite set, Q is a cocartesian *𝒮et*-endofunctor, and Σ and Δ are polynomial. ◇

In order to describe arbitrary macro tree transducers we need the adjunction $\mathsf{F} \dashv |\,.\,|$ where F is the free-functor $\mathsf{F} : \mathbb{T}_\Sigma \leftarrow\!\dashv \Sigma$ mapping a varietor onto its free monad.

6.2.8 Proposition. The macro tree transducers are precisely the monadic transducers

$$\mathbb{M} = \left(\overline{\mathsf{F} \cdot \mathsf{A}_I \cdot \mathsf{Q}} \cdot (\,.\,)^+, \Sigma, \Delta, \omega, \varrho\right)$$

over *𝒮et* where I is a finite set, Q is a cocartesian *𝒮et*-endofunctor, and Σ and Δ are polynomial. ◇

7 Fusion of tree transducers

In Section 6 we have seen how particular functional programs can be equivalently transformed into monadic transducers. Now we are ready to apply the monadic fusion Theorem 5.2.2 to functional programs:

7.1 Fusion of individual tree transducers

Given algebraic types A, B, and C and functional programs $C \xleftarrow{g} B \xleftarrow{f} A$ we construct a new program $C \xleftarrow{h} A$ such that

$$\begin{tikzcd}
\mathsf{T}_\Gamma\emptyset & \mathsf{T}_\Delta\emptyset \arrow[l, "{[g]}"'] & \mathsf{T}_\Sigma\emptyset \arrow[l, "{[f]}"'] \arrow[ll, bend left, "{[\![h]\!]}"]
\end{tikzcd}$$

where the initial term-algebras T_Σ, T_Δ, andT_Γ are supposed to be the semantics of the types A, B, and C, respectively. The construction of h is shown in Figure 4. The rule of h is constructed according to Definition 5.2.1. The correctness of the fusion transformation w.r.t. the denotational semantics $[\![\,.\,]\!]$ follows from Theorem 5.2.2. Notice, that there occurs no Δ in the rule of the fusion, and thus the intermediate data structure has indeed been removed.

This algorithm works for all functional programs which can be written as monadic transducers (and more as outlined in Subsection 8.1). In particular it works for the classes of tree transducers from Section 6.

7.2 Fusion of classes of tree transducers

Our new fusion theorem makes it possible to fuse classes of tree transducers (or more precisely: classes of functions computable by a class of tree transducers). We have already seen the following classes: $HOM \subsetneq TOP \subsetneq sb\text{-}MAC \subsetneq b\text{-}MAC \subsetneq MAC$. From the theory of tree transducer it is known that all the above inclusions are proper.

Let ID be the class of all identity functions. For all classes of functions A and B we define the composition

$$A \cdot B = \left\{ a \cdot b \;\middle|\; a \in A \wedge b \in B \wedge \operatorname{dom} a = \operatorname{cod} b \right\}.$$

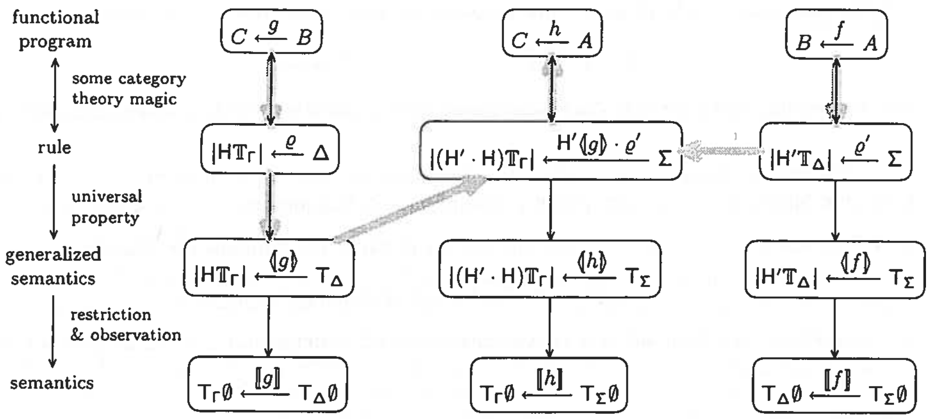

functional program
some category theory magic
rule
universal property
generalized semantics
restriction & observation
semantics
$C \xleftarrow{g} B$
$C \xleftarrow{h} A$
$B \xleftarrow{f} A$
$|\mathsf{H}\mathbb{T}_\Gamma| \xleftarrow{\varrho} \Delta$
$|(\mathsf{H}' \cdot \mathsf{H})\mathbb{T}_\Gamma| \xleftarrow{\mathsf{H}'(\!|g|\!) \cdot \varrho'} \Sigma$
$|\mathsf{H}'\mathbb{T}_\Delta| \xleftarrow{\varrho'} \Sigma$
$|\mathsf{H}\mathbb{T}_\Gamma| \xleftarrow{(\!|g|\!)} \mathsf{T}_\Delta$
$|(\mathsf{H}' \cdot \mathsf{H})\mathbb{T}_\Gamma| \xleftarrow{(\!|h|\!)} \mathsf{T}_\Sigma$
$|\mathsf{H}'\mathbb{T}_\Delta| \xleftarrow{(\!|f|\!)} \mathsf{T}_\Sigma$
$\mathsf{T}_\Gamma\emptyset \xleftarrow{[\![g]\!]} \mathsf{T}_\Delta\emptyset$
$\mathsf{T}_\Gamma\emptyset \xleftarrow{[\![h]\!]} \mathsf{T}_\Sigma\emptyset$
$\mathsf{T}_\Delta\emptyset \xleftarrow{[\![f]\!]} \mathsf{T}_\Sigma\emptyset$

Figure 4: Monadic fusion algorithm

Obviously $ID \subseteq HOM$ because $[\![\mathbb{ID}_\Sigma]\!] = id_{\mathsf{T}_\Sigma}$. Moreover, if $ID \subseteq B$ then $A \subseteq A \cdot B \cap B \cdot A$.

According to Definition 5.1.1 and Theorem 5.2.2 it suffices to compose the patterns of the according monadic transducers from Section 6 in order to compose two of the classes HOM, TOP, $sb\text{-}MAC$, $b\text{-}MAC$, or MAC.

It is obvious that all classes of monadic transducers are closed under composition with HOM from either side, because the pattern of a homomorphism transducer is the identity functor Id.

7.2.1 Theorem (fusion of top-down tree transducers [Eng75]**).**

$$TOP \cdot TOP = TOP. \qquad \diamond$$

Proof. Let $\overline{\mathsf{Q}}$ and $\overline{\mathsf{Q}'}$ be the pattern of the two top-down tree transducers where Q and Q' are cocartesian $\mathcal{S}et$-endofunctors according to Proposition 6.2.3. We just have to compose the patterns and use Lemma 4.3.5: $\overline{\mathsf{Q}'} \cdot \overline{\mathsf{Q}} = \overline{\mathsf{Q} \cdot \mathsf{Q}'}$. Since cocartesian functors are closed under composition the latter is again the pattern of a top-down tree transducer. ∎

7.2.2 Theorem (fusion of a ((simple) basic) macro and a top-down tree transducer [Eng81]**).**

$$\begin{aligned} sb\text{-}MAC \cdot TOP &= sb\text{-}MAC \\ b\text{-}MAC \cdot TOP &= b\text{-}MAC \\ MAC \cdot TOP &= MAC. \end{aligned}$$

Proof. The pattern of a top-down tree transducer is $\overline{\mathsf{Q}'}$ and the pattern of any of the above macro tree transducers has the form $\overline{\mathsf{Q} \cdots}$ where Q and Q' are cocartesian $\mathcal{S}et$-endofunctors according to the Propositions 6.2.3, 6.2.6, 6.2.7, and 6.2.8. As in the proof of Theorem 7.2.1 we calculate the pattern of the fusion: $\overline{\mathsf{Q} \cdot \mathsf{Q}' \cdots}$. ∎

The following lemmas help us to calculate with patterns of monadic transducers:

7.2.3 Lemma. Let $\mathcal{C}$ be a category which has function spaces, I be a $\mathcal{C}$ object, and $\mathsf{Q} \dashv \mathsf{U} : \mathcal{C} \leftarrow \mathcal{C}$. Then the following holds:

$$\mathsf{A}_{\mathsf{Q}I} \cong (\mathsf{Id} \leftarrow\!\!\circ\ \mathsf{U}) \cdot \mathsf{A}_I. \qquad \diamond$$

7.2.4 Lemma. Let $\mathcal{C}$ be a bicartesian closed category, I be a $\mathcal{C}$-object, and Q a cocartesian $\mathcal{C}$-endofunctor. Then the following holds:

$$\mathsf{A}_I \cdot \mathsf{Q} \cong (\mathsf{Q} \leftarrow\!\!\circ\, \mathsf{Id}) \cdot \mathsf{A}_I. \qquad \diamond$$

7.2.5 Corollary. Let $\mathcal{C}$ be a bicartesian closed category which has function spaces, I be a $\mathcal{C}$ object, and Q a cocartesian $\mathcal{C}$-endofunctor with right adjoint U. Putting together the Lemmas 7.2.3 and 7.2.4 and with a little help from Definition 4.3.2, Lemmas 4.3.6 and 6.2.4 we get:

$$\overline{\mathsf{Q}} \cdot \overline{\mathsf{A}_{\mathsf{Q}I}} \cong \overline{\mathsf{A}_I} \cdot \overline{\mathsf{Q} \leftarrow\!\!\circ\, \mathsf{U}}. \qquad \diamond$$

7.2.6 Lemma. Let $\mathsf{Q} \dashv \mathsf{U} : \mathcal{C} \leftarrow \mathcal{D}$ be an adjunction and $\mathbb{T}$ and $\mathbb{T}'$ be monads on $\mathcal{C}$. Then:

$$\overline{\mathsf{Q} \leftarrow\!\!\circ\, \mathsf{U}}(\mathbb{T} \leftarrow\!\!\circ\, \mathbb{T}') \cong \overline{\mathsf{Q}}\mathbb{T} \leftarrow\!\!\circ\, \overline{\mathsf{Q}}\mathbb{T}'. \qquad \diamond$$

For the following theorem we will need Corollary 4.4.4, Corollary 7.2.5, and Lemma 7.2.6 to compose the patterns. Then we have to use Lemma 4.3.8 and Corollary 4.4.4 to construct a monadic transducer homomorphism from the composed pattern to the wanted pattern of the fusion (as they are not necessarily equal). Finally we use Theorem 5.3.2 to show the equivalence of the two semantics and thus of the respective classes of tree transducers:

7.2.7 Theorem (fusion of a top-down and a ((simple) basic) macro tree transducer [EV85]**).**

$$\begin{aligned} TOP \cdot sb\text{-}MAC &= sb\text{-}MAC \\ TOP \cdot b\text{-}MAC &= b\text{-}MAC \\ TOP \cdot MAC &= MAC. \end{aligned} \qquad \diamond$$

8 Generalizations and future work

Finally we outline some topics for further research:

The monadic fusion theorem (Theorem 5.2.2) works in arbitrary categories (which have an initial object). We want to investigate applications of the monadic fusion in other categories than *Set*, *e.g.*: *bottom-up tree transducers* or functional programs with infinite data structures.

The monadic fusion guarantees the elimination of the intermediate data structure. However, this does not necessarily imply that the resulting program will be more efficient. We want to compare the efficency of the programs before and after the monadic fusion.

We would like to extend our monadic fusion to *tree to tree-series transducers* which will be possible with the following generalization:

8.1 Using arbitrary monads

Consider the rule $\varrho : |\mathsf{H}\mathbb{T}_\Delta| \dot{\leftarrow} \Sigma$ of a monadic transducers according to Definition 5.1.1. It is easy to see that (i) and (ii) of Theorem 5.2.2 make no use of the fact that $\mathbb{T}_\Delta$ is a *free* monad over Δ. Hence we can define a generalized version of a monadic transducer where the rule has the form $\varrho : |\mathsf{H}\mathbb{T}| \dot{\leftarrow} \Sigma$ where $\mathbb{T}$ is an *arbitrary* monad. Then we can still apply our fusion theorem (Theorem 5.2.2 (ii)) even if the consuming monadic transducer is generalized.

References

[AHS90] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Pure and Applied Mathematics. John Wiley & Sons, 1990.

[CDPR97a] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Attribute grammars and functional programming deforestation. In *4th International Static Analysis Symposium—Poster Session*, Paris (F), 1997.

[CDPR97b] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1997.

[CF82] B. Courcelle and P. Franchi–Zannettacci. Attribute grammars and recursive program schemes. *Theoret. Comput. Sci.*, 17:163–191, 235–257, 1982.

[Dam82] W. Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20:95–206, 1982.

[Eng75] J. Engelfriet. Bottom-up and top-down tree transformations—a comparison. *Math. Systems Theory*, 9(3):198–231, 1975.

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal language theory: perspectives and open problems*, pages 241–286. New York, Academic Press, 1980.

[Eng81] J. Engelfriet. Tree transducers and syntax-directed semantics. Technical Report Memorandum 363, Technische Hogeschool Twente, March 1981. also in: Proceedings of the Colloquium on Trees in Algebra and Programming (CAAP'92), Lille, France 1992.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. System Sci.*, 31:71–146, 1985.

[Fok92] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[Fül81] Z. Fülöp. On attributed tree transducers. *Acta Cybernet.*, 5:261–279, 1981.

[FV98] Z. Fülöp and H. Vogler. *Syntax-directed semantics—Formal models based on tree transducers*. Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1998.

[Gie88] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Inform.*, 25:355–423, 1988.

[Gil96] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, Glasgow University, January 1996.

[GLP93] A. Gill, J. Launchburry, and S. L. Peyton-Jones. A short cut to deforestation. In *Proceedings of Functional Programming Languages an Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

[Hin00] R. Hinze. Deriving backtracking monad transformers. In P. Wadler, editor, *Proceedings of the 2000 International Conference on Functional Programming (ICFP'03)*, Montreal, Canada, sep 2000.

[Joh01] P. Johann. Short cut fusion: Proved and improved. In W. Taha, editor, *Proceedings of the 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 47–71, Florence, Italy, September 2001. Springer.

[Jür02] C. Jürgensen. Monadic fusion of functional programs. Technical Report TUD-FI02-12, Technische Universität Dresden, Fakultät Informatik, D-01062 Dresden, Germany, December 2002.

[JV01] C. Jürgensen and H. Vogler. Syntactic composition of top-down tree transducers is short cut fusion. Technical Report TUD-FI01-10, Technische Universität Dresden, Fakultät Informatik, D-01062 Dresden, Germany, November 2001. *Accepted for publication in Math. Struct. in Comp. Science.*

[Kel80] G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bulletins of the Australian Mathematical Society*, 22:1–83, 1980.

[Küh98] A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and R. Ramanujam, editors, *Proceedings of the 18th INternational Conference on Foundations of Software Technology & Theoretical Computer Science (FST&TCS'98)*, volume 1530 of *LNCS*, pages 146–157, Chennai, India, dec 1998. Springer-Verlag.

[KV01] A. Kühnemann and J. Voigtländer. Tree transducer composition as deforestation method for functional programs. Technical Report TUD-FI01-07, Technische Universität Dresden, Fakultät Informatik, D-01062 Dresden, Germany, August 2001.

[LG02a] Ch. Lüth and N. Ghani. Composing monads using coproducts. In *International Conference on Functional Programming (ICFP'02)*, pages 133– 144. ACM Press, September 2002.

[LG02b] Ch. Lüth and N. Ghani. Monads and modularity. In Alessandro Armando, editor, *Frontiers of Combining Systems FroCos 2002, 4th International Workshop*, number 2309 in Lecture Notes in Artificial Intelligence, pages 18–32. Springer Verlag, 2002.

[Mog90] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, 1990.

[Rou68] W. C. Rounds. *Trees, transducers and transformations*. PhD thesis, Stanford University, 1968.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Conference on Functional Programing Languages and Computer Architecture*, pages 306–313, La Jolla, CA, June 1995. ACM Press.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

An Abstract Monadic Semantics for Value Recursion

Eugenio Moggi[*]
DISI
Univ. di Genova
moggi@disi.unige.it

Amr Sabry[†]
Dept. of Computer Science
Indiana University
sabry@indiana.edu

Abstract

This paper proposes an operational semantics for value recursion in the context of monadic metalanguages. Our technique for combining value recursion with computational effects works *uniformly* for all monads. The operational nature of our approach is related to the implementation of recursion in Scheme and its monadic version proposed by Friedman and Sabry, but it defines a different semantics and does not rely on assignments. When contrasted to the axiomatic approach proposed by Erkök and Launchbury, our semantics for the continuation monad invalidates one of the axioms, adding to the evidence that this axiom is problematic in the presence of continuations.

1 Introduction

How should recursive definitions interact with computational effects like assignments and jumps? Consider a term *fix* x.e where *fix* is some fixed point operator and e is an expression whose evaluation has side-effects. There are at least two natural meanings for the term:

1. the term is equivalent to the unfolding $e\{x = \mathit{fix}\ x.e\}\Gamma$and the side-effects are duplicated by the unfolding.
2. the side-effects are performed the first time e is evaluated to a value v and then the term becomes equivalent to the unfolding $v\{x = \mathit{fix}\, x.v\}$.

The first meaning corresponds to the standard mathematical view [Bar84]. The second meaning corresponds to the standard operational view defined

[*]Supported by MIUR project NAPOLI and EU project DART IST-2001-33477.

[†]This material is based upon work supported by the NSF under Grants No. CCR 0196063 and CCR 0204389.

since the SECD machine [Lan64] and as implemented in Scheme for example [KCE98]. The two meanings are observationally equivalent in a pure functional language. When the computational effects are expressed using monadsΓErkök and Launchbury [Erk02ΓEL00ΓELM02] introduced the phrase *value recursion in monadic computations* for the second meaning and the name *mfix* for the corresponding fixed-point operator. Since we also work in the context of monadic metalanguagesΓwe adopt the same terminology but use the capitalized name *Mfix* to distinguish our approach.

We propose a simple uniform operational technique for combining monadic effects with value recursion. Computing the result of *Mfix* $x.e$ requires three rules:

1. A rule to initiate the computation of e. Since this computation happens under a binderΓcare must be taken to rename any other bound instance of x that we might later encounter.
2. If the computation of e returns a value vΓall free occurrences of x are replaced by *fix* $x.v$ (where *fix* is the standard mathematical fixed-point operator).
3. If the computation of e attempts to use xΓwe signal an error.

The three rules above are robust in the sense that they can be uniformly applied to a wide range of monads: we give examples for the monads of stateΓnon-determinismΓparallelismΓand continuations.

Our semantics is operational in nature but unlike the SECD and Scheme semanticsΓit doesn't rely on assignments to realize the second rule. The presence of assignments in the other operational approaches yields a different semanticsΓcomplicates reasoningΓand invalidates some equational axioms.

In contrastΓthe work by Erkök and Launchbury [EL00ΓErk02] advocates an axiomatic approach to defining value recursion by proposing several desirable axioms. In their approach one has to find for each given monad over some category (or defined in Haskell [Jon99]) a fixed point operator that satisfy the axioms (up to observational equivalence). The endeavor has to be repeated for each monad individually. For the continuation monad there are no known fixed point operators that satisfy all the desired axioms.

Summary. Sections 2 and 3 illustrate the technique by taking an existing monadic metalanguage MML^S with ML-style references [MF03ΓSec.3] and extending it with value recursion. Section 4 recalls the equational axioms for value recursion in [Erk02]Γand when they are known to fail. Section 5 shows that the addition of value recursion to MML^S is robust with respect to the addition of other computational effectsΓnamely non-determinism and parallelism. FinallyΓSection 6 explains the full subtleties of value recursion in the presence of continuationsΓoutlines a proof of type safetyΓand discusses counter-examples to equational axioms.

2 A Monadic Metalanguage with References

We introduce a monadic metalanguage MML^S for imperative computationsΓ namely a subset of Haskell with the IO-monad. Its operational semantics is given according to the general pattern proposed in [MF03]Γi.e. we specify a confluent *simplification* relation $\longrightarrow$ (defined as the *compatible closure* of a set of rewrite rules)Γand a *computation* relation $\longmapsto$ describing how the *configurations* of the (closed) system may evolve. This is possible because in a monadic metalanguage there is a clear distinction between term-constructors for building terms of computational typesΓand the other term-constructors that are *computationally irrelevant* (i.e. have no effects). For computationally relevant term-constructors we give an operational semantics that ensures the correct sequencing of computational effectsΓe.g. by adopting some well-established technique for specifying the operational semantics of programming languages (see [WF94])Γwhile for computationally irrelevant term-constructors it suffices to give local simplification rulesΓthat can be applied non-deterministically (because they are semantic preserving).
The syntax of MML^S is abstracted over basic types bΓvariables $x \in \mathsf{X}$Γand locations $l \in \mathsf{L}$.

- Types $\boxed{\tau \in \mathsf{T} ::= b \mid \tau_1 \rightarrow \tau_2 \mid M\tau \mid R\tau}$
- Terms $\boxed{\begin{array}{lll} e \in \mathsf{E} & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid \mathit{ret}\ e \mid \mathit{do}\ x \leftarrow e_1; e_2 \mid \\ & & l \mid \mathit{new}\ e \mid \mathit{get}\ e \mid \mathit{set}\ e_1\ e_2 \end{array}}$

In addition to the basic typesΓwe have function types $\tau_1 \rightarrow \tau_2$Γreference types $R\tau$ for locations containing values of type τΓand computational types $M\tau$ for (effect-full) programs computing values of type τ. The terms $\mathit{do}\ x \leftarrow e_1; e_2$ and $\mathit{ret}\ e$ are used to sequence and terminate computationsΓ the other monadic operations are: $\mathit{new}\, e$ which creates a new referenceΓ$\mathit{get}\ e$ which returns the contents of a referenceΓand $\mathit{set}\ e_1\ e_2$ which updates the contents of reference e_1 to be e_2. In order to specify the semantics of the languageΓthe set of terms also includes locations l.
Table 1 gives the typing rules (for deriving judgments of the form $\Gamma \vdash_\Sigma e{:}\,\tau$Γ where $\Gamma{:}\,\mathsf{X} \overset{fin}{\rightarrow} \mathsf{T}$ is a type assignment for variables $x{:}\,\tau$ and $\Sigma{:}\,\mathsf{L} \overset{fin}{\rightarrow} \mathsf{T}$ is a signature for locations $l{:}\,R\tau$.
The operational semantics is given by two relations (as outlined above): a *simplification* relation for pure evaluation and a *computation* relation for monadic evaluation. Simplification $\longrightarrow$ is given by β-reductionΓi.e. the compatible closure of $(\lambda x.e_2)e_1 \longrightarrow e_2\{x{:}= e_1\}$. The computation relation

$$x\ \frac{\Gamma(x)=\tau}{\Gamma\vdash_\Sigma x:\tau} \qquad \text{abs}\ \frac{\Gamma,x:\tau_1\vdash_\Sigma e:\tau_2}{\Gamma\vdash_\Sigma \lambda x.e:\tau_1\to\tau_2}$$

$$\text{app}\ \frac{\Gamma\vdash_\Sigma e_1:\tau_1\to\tau_2 \quad \Gamma\vdash_\Sigma e_2:\tau_1}{\Gamma\vdash_\Sigma e_1 e_2:\tau_2}$$

$$\text{ret}\ \frac{\Gamma\vdash_\Sigma e:\tau}{\Gamma\vdash_\Sigma \mathit{ret}\ e: M\tau} \qquad \text{do}\ \frac{\Gamma\vdash_\Sigma e_1: M\tau_1 \quad \Gamma,x:\tau_1\vdash_\Sigma e_2: M\tau_2}{\Gamma\vdash_\Sigma \mathit{do}\ x\leftarrow e_1; e_2: M\tau_2}$$

$$l\ \frac{\Sigma(l)=R\tau}{\Gamma\vdash_\Sigma l: R\tau} \qquad \text{new}\ \frac{\Gamma\vdash_\Sigma e:\tau}{\Gamma\vdash_\Sigma \mathit{new}\ e: M(R\tau)} \qquad \text{get}\ \frac{\Gamma\vdash_\Sigma e: R\tau}{\Gamma\vdash_\Sigma \mathit{get}\ e: M\tau}$$

$$\text{set}\ \frac{\Gamma\vdash_\Sigma e_1: R\tau \quad \Gamma\vdash_\Sigma e_2:\tau}{\Gamma\vdash_\Sigma \mathit{set}\ e_1\ e_2: M(R\tau)}$$

Table 1: Type System for MML^S

$Id \longmapsto Id' \mid$ done (see Table 2) is defined using the additional notions of evaluation contextsΓstores and configurations $Id \in \mathsf{Conf}$:

- Evaluation contexts $\boxed{E \in \mathsf{EC} ::= \Box \mid E[\mathit{do}\ x \leftarrow \Box; e]}$ (or equivalently $E ::= \Box \mid \mathit{do}\ x \leftarrow E; e$).
- Stores $\mu \in \mathsf{S} \triangleq \mathsf{L} \overset{fin}{\to} \mathsf{E}$ map locations to their contents.
- Configurations $(\mu, e, E) \in \mathsf{Conf} \triangleq \mathsf{S}\times\mathsf{E}\times\mathsf{EC}$ consist of the current store μΓthe program fragment e under considerationΓand its evaluation context E.

3 Extension with Value Recursion

We now describe the monadic metalanguage $\mathsf{MML}^S_{\mathit{fix}}$ obtained by extending MML^S with two fixed point constructs: *fix* $x.e$ for ordinary recursionΓ and *Mfix* $x.e$ for value recursion. The expression *fix* $x.e$ *simplifies* to its unfolding. For *computing* the value of *Mfix* $x.e$Γthe subexpression e is first evaluated to a monadic value *ret* e'. This evaluation might perform computational effects but cannot use x. Then all occurrences of x in e' are bound to the monadic value itself using *fix* so that any unfolding will not redo the computational effects.

The extension $\mathsf{MML}^S_{\mathit{fix}}$ is an *instance* of a general pattern (only the extension of the computation relation is non-trivial)Γthat will become clearer after considering other monadic metalanguages.

Administrative steps

(A.0) $(\mu, \mathit{ret}\ e, \Box) \longmapsto \mathsf{done}$

(A.1) $(\mu, \mathit{do}\ x \leftarrow e_1; e_2, E) \longmapsto (\mu, e_1, E[\mathit{do}\ x \leftarrow \Box; e_2])$

(A.2) $(\mu, \mathit{ret}\ e_1, E[\mathit{do}\ x \leftarrow \Box; e_2]) \longmapsto (\mu, e_2\{x := e_1\}, E)$

Imperative steps

(new) $(\mu, \mathit{new}\ e, E) \longmapsto (\mu\{l: e\}, \mathit{ret}\ l, E)$ where $l \notin \mathsf{dom}(\mu)$

(get) $(\mu, \mathit{get}\ l, E) \longmapsto (\mu, \mathit{ret}\ e, E)$ with $e = \mu(l)$

(set) $(\mu, \mathit{set}\ l\ e, E) \longmapsto (\mu\{l = e\}, \mathit{ret}\ l, E)$ with $l \in \mathsf{dom}(\mu)$

Table 2: Computation Relation for MML^S

- Terms $\boxed{e \in \mathsf{E} \mathrel{+}= \mathit{fix}\ x.e \mid \mathit{Mfix}\ x.e}$
- Evaluation contexts $\boxed{E \in \mathsf{EC} \mathrel{+}= E[\mathit{Mfix}\ x.\Box]}$
- Configurations $(X|\mu, e, E) \in \mathsf{Conf} \triangleq \mathcal{P}_{\mathit{fin}}(\mathsf{X}) \times \mathsf{S} \times \mathsf{E} \times \mathsf{EC}$. The additional component X is a set which records the recursive variables generated so farΓthus X grows as the computation progresses.

Despite their different semanticsΓthe two fixed points have similar typing rules:

$$\frac{\Gamma, x: M\tau \vdash_\Sigma e: M\tau}{\Gamma \vdash_\Sigma \mathit{fix}\ x.e: M\tau} \qquad \frac{\Gamma, x: M\tau \vdash_\Sigma e: M\tau}{\Gamma \vdash_\Sigma \mathit{Mfix}\ x.e: M\tau}$$

The simplification relation is extended with the rule $\mathit{fix}\ x.e \longrightarrow e\{x := \mathit{fix}\ x.e\}$ for *fix*-unfolding.

The computation relation $\mathit{Id} \longmapsto \mathit{Id}' \mid \mathsf{done} \mid \mathsf{err}$ may now raise an error and is defined by the following rules:

- the rules in Table 2Γmodified to propagate the set X unchangedΓand
- the following new rules for evaluating recursive bindings $\mathit{Mfix}\ x.e$:

(M.1) $(X|\mu, \mathit{Mfix}\ x.e, E) \longmapsto (X, x|\mu, e, E[\mathit{Mfix}\ x.\Box])$ with x renamed to avoid clashes with X

(M.2) $(X|\mu, \mathit{ret}\ e, E[\mathit{Mfix}\ x.\Box]) \longmapsto (X|\overline{\mu}, \mathit{ret}\ \overline{e}, \overline{E})$ where $\overline{\bullet}$ stands for $\bullet\{x := \mathit{fix}\ x.\mathit{ret}\ e\}$

(err) $(X|\mu, x, E) \longmapsto$ err where $x \in X$ (attempt to use an unresolved recursive variable)

In the context *Mfix* $x.\Box$ the hole is within the scope of a binderΓthus it requires evaluation of open terms:

- The rule (M.1) behaves like *gensym*Γit ensures *freshness* of x. As the computation progresses x may leak anywhere in the configuration (depending on the computational effects available in the language).
- The rule (M.2) does the reverseΓit replaces all *free occurrences* of x in the configuration with the term *fix* x.*ret* eΓin which x is not free. This rule is quite subtleΓbecause of $E\{x := e\}$ (see Definition 6.5).

In special cases [AFMZ02] it is possible to simplify (M.2) by treating X as a stack and enforcing the invariant that $\mathrm{FV}(E) = \emptyset$Γbut our aim is an operational semantics that works with *arbitrary* computational effects. Indeed in the case of continuations (Section 6)Γneither of these invariants holds.

4 Axioms for Value Recursion

In [Erk02] the fixed point constructs have a slightly different typing:

- $\dfrac{\Gamma, x{:}\,\tau \vdash_\Sigma e{:}\, M\tau}{\Gamma \vdash_\Sigma \mathit{mfix}\ x.e{:}\, M\tau}$ where x is of type τ.

  This rule allows the use of x at type τ before the recursion is resolvedΓ as in $(\mathit{mfix}\ x.\mathit{set}\ x\ 0){:}\, M(R\ \mathit{int})$. In [Erk02] this premature attempt to use x is identified with *divergence*Γwhile but we consider it a *monadic error*Γwhich should be statically prevented by more refined type systems [Bou01]. The difference of typing reflects this desire and is not an intrinsic limitation of our approach.

- $\dfrac{\Gamma, x{:}\,\tau \vdash_\Sigma e{:}\, \tau}{\Gamma \vdash_\Sigma \mathit{fix}\ x.e{:}\, \tau}$ requires recursive definitions at *all types*; we only require them at *computational types*.

Two of the important axioms for defining value recursion in [Erk02] are:

$$\begin{array}{lrcl} \text{(Purity)} & \mathit{mfix}\ x.\mathit{ret}\ e & = & \mathit{ret}\ (\mathit{fix}\ x.e) \\ \text{(Left-shrinking)} & \mathit{mfix}\ x.(\mathit{do}\ x_1 \leftarrow e_1; e_2) & = & \mathit{do}\ x_1 \leftarrow e_1; \mathit{mfix}\ x.e_2 \\ & & & \text{when } x \notin \mathrm{FV}(e_1) \end{array}$$

The *purity* axiom requires that *mfix* coincides with *fix* for pure computations. Because of the differences in typingΓthe *purity* axiom in our case becomes:

$$\text{(Purity)} \qquad \mathit{Mfix}\ x.\mathit{ret}\ e \;=\; \mathit{fix}\ x.\mathit{ret}\ e$$

Left-shrinking states that computations which do not refer to the recursive variable can be moved outside the recursive definition. This rewriting however is known to be incorrect in Scheme [Baw88] but it was argued [Erk02] that the failure of left-shrinking is due to the idiosyncrasies of Scheme. In fact left-shrinking is invalidated by our semantics and in other known combinations of value recursion and continuations [FS00ΓCar03]. Indeed if one captures the continuation in e_1 then on the left-hand side this continuation has access to free occurrences of x in e_2 but not on the right-hand side. As Section 6.2 illustrates this can be exploited to write a counterexample to left-shrinking.

5 Non-Determinism and Parallelism

We consider two extensions to MML^S (and $\mathsf{MML}^S_{\mathit{fix}}$): the first introduces non-deterministic choice e_1 *or* e_2Γthe second introduces a construct *spawn* e_1 e_2 to spawn a thread of computation e_1 in parallel with the continuation e_2 of the current thread.

Non-determinism. The typing rule for non-deterministic choice is:

$$\frac{\Gamma \vdash_\Sigma e_1 \colon M\tau \quad \Gamma \vdash_\Sigma e_2 \colon M\tau}{\Gamma \vdash_\Sigma e_1 \ \mathit{or}\ e_2 \colon M\tau}$$

The configurations for MML^S and $\mathsf{MML}^S_{\mathit{fix}}$ are unchanged. The computation relations are modified to become non-deterministic. More specificallyΓ

- for MML^SΓwe add the computation rules $(\mu, e_1 \ \mathit{or}\ e_2, E) \longmapsto (\mu, e_i, E)$ for $i = 1, 2$;
- for $\mathsf{MML}^S_{\mathit{fix}}$Γwe add the rules $(X|\mu, e_1 \ \mathit{or}\ e_2, E) \longmapsto (X|\mu, e_i, E)$ for $i = 1, 2$.

Parallelism. The typing rule for *spawn* is:

$$\frac{\Gamma \vdash_\Sigma e_1 \colon M\tau_1 \quad \Gamma \vdash_\Sigma e_2 \colon M\tau_2}{\Gamma \vdash_\Sigma \mathit{spawn}\ e_1\ e_2 \colon M\tau_2}$$

In this case a configuration consists of a (finite) multi-set of parallel threads sharing the store μΓwhere each thread is represented by a pair (e, E).

For MML^S the configurations become $\langle \mu, N \rangle \in \mathsf{Conf} \stackrel{\Delta}{=} \mathsf{S} \times \mathcal{M}_{\mathit{fin}}(\mathsf{E} \times \mathsf{EC})$Γi.e. instead of a thread (e, E) one has a multi-set of threadsΓand the computation relation $Id \longmapsto Id' \mid \mathsf{done}$ is defined by the following rules:

- Administrative steps: threads act independentlyΓtermination occurs when all threads have completed

(done) $\langle \mu, \emptyset \rangle \longmapsto \mathsf{done}$

(A.0) $\langle \mu, (\mathit{ret}\ e, \square) \uplus N \rangle \longmapsto \langle \mu, N \rangle$

(A.1) $\langle \mu, (\mathit{do}\ x \leftarrow e_1; e_2, E) \uplus N \rangle \longmapsto \langle \mu, (e_1, E[\mathit{do}\ x \leftarrow \square; e_2]) \uplus N \rangle$

(A.2) $\langle \mu, (\mathit{ret}\ e_1, E[\mathit{do}\ x \leftarrow \square; e_2]) \uplus N \rangle \longmapsto \langle \mu, (e_2\{x := e_1\}, E) \uplus N \rangle$

- Imperative steps: each thread can operate on the shared store

(new) $\langle \mu, (\mathit{new}\ e, E) \uplus N \rangle \longmapsto \langle \mu\{l : e\}, (\mathit{ret}\ l, E) \uplus N \rangle$ where $l \notin \mathsf{dom}(\mu)$

(get) $\langle \mu, (\mathit{get}\ l, E) \uplus N \rangle \longmapsto \langle \mu, (\mathit{ret}\ e, E) \uplus N \rangle$ with $e = \mu(l)$

(set) $\langle \mu, (\mathit{set}\ l\ e, E) \uplus N \rangle \longmapsto \langle \mu\{l = e\}, (\mathit{ret}\ l, E) \uplus N \rangle$ with $l \in \mathsf{dom}(\mu)$

- Step for spawning a new thread

(spawn) $\langle \mu, (\mathit{spawn}\ e_1\ e_2, E) \uplus N \rangle \longmapsto \langle \mu, (e_1, \square) \uplus (e_2, E) \uplus N \rangle$

For $\mathsf{MML}^{S}_{\mathit{fix}}$ the configurations become $\langle X|\mu, N \rangle \in \mathsf{Conf} \triangleq \mathcal{P}_{\mathit{fin}}(\mathsf{X}) \times \mathsf{S} \times \mu(\mathsf{E} \times \mathsf{EC})$Γi.e. the threads share the set X which records the recursive variables generated so farΓand the computation relation $Id \longmapsto Id' \mid \mathsf{done} \mid \mathsf{err}$ is defined by the rules above (modified to propagate the set X unchanged) and the following rules for recursive monadic bindings:

(M.1) $\langle X|\mu, (\mathit{Mfix}\ x.e, E) \uplus N \rangle \longmapsto \langle X, x|\mu, (e, E[\mathit{Mfix}\ x.\square]) \uplus N \rangle$ with x renamed to avoid clashes with X

(M.2) $\langle X|\mu, (\mathit{ret}\ e, E[\mathit{Mfix}\ x.\square]) \uplus N \rangle \longmapsto \langle X|\overline{\mu}, (\mathit{ret}\ \overline{e}, \overline{E}) \uplus \overline{N} \rangle$ where $\overline{\bullet}$ stands for $\bullet\{x := \mathit{fix}\ x.\mathit{ret}\ e\}$

(err) $\langle X|\mu, (x, E) \uplus N \rangle \longmapsto \mathsf{err}$ where $x \in X$

When a thread resolves a recursive variable x (M.2)Γthe value of x is propagated to all other threads. When an error occurs in a thread (err)Γthe whole computation crashes.

6 References and Continuations

In this section we consider in full detail the monadic metalanguage $\mathsf{MML}^{SK}_{\mathit{fix}}$Γ obtained from $\mathsf{MML}^{S}_{\mathit{fix}}$ by adding continuations. Section 6.1 outlines a proof of type safetyΓand Section 6.2 shows the failure of the left-shrinking axiom and discusses some differences with Scheme. The syntax of $\mathsf{MML}^{SK}_{\mathit{fix}}$ is abstracted over basic types bΓvariables $x \in \mathsf{X}$Γlocations $l \in \mathsf{L}$ and continuations $k \in \mathsf{K}$:

$$x\ \frac{\Gamma(x)=\tau}{\Gamma\vdash_\Sigma x:\tau}\qquad \text{abs}\ \frac{\Gamma,x:\tau_1\vdash_\Sigma e:\tau_2}{\Gamma\vdash_\Sigma \lambda x.e:\tau_1\to\tau_2}$$

$$\text{app}\ \frac{\Gamma\vdash_\Sigma e_1:\tau_1\to\tau_2\quad \Gamma\vdash_\Sigma e_2:\tau_1}{\Gamma\vdash_\Sigma e_1e_2:\tau_2}\qquad \text{fix}\ \frac{\Gamma,x:M\tau\vdash_\Sigma e:M\tau}{\Gamma\vdash_\Sigma \mathit{fix}\,x.e:M\tau}$$

$$\text{ret}\ \frac{\Gamma\vdash_\Sigma e:\tau}{\Gamma\vdash_\Sigma \mathit{ret}\ e:M\tau}\qquad \text{do}\ \frac{\Gamma\vdash_\Sigma e_1:M\tau_1\quad \Gamma,x:\tau_1\vdash_\Sigma e_2:M\tau_2}{\Gamma\vdash_\Sigma \mathit{do}\ x\leftarrow e_1;e_2:M\tau_2}$$

$$\text{Mfix}\ \frac{\Gamma,x:M\tau\vdash_\Sigma e:M\tau}{\Gamma\vdash_\Sigma \mathit{Mfix}\,x.e:M\tau}$$

$$l\ \frac{\Sigma(l)=R\tau}{\Gamma\vdash_\Sigma l:R\tau}\qquad \text{new}\ \frac{\Gamma\vdash_\Sigma e:\tau}{\Gamma\vdash_\Sigma \mathit{new}\ e:M(R\tau)}\qquad \text{get}\ \frac{\Gamma\vdash_\Sigma e:R\tau}{\Gamma\vdash_\Sigma \mathit{get}\ e:M\tau}$$

$$\text{set}\ \frac{\Gamma\vdash_\Sigma e_1:R\tau\quad \Gamma\vdash_\Sigma e_2:\tau}{\Gamma\vdash_\Sigma \mathit{set}\ e_1\ e_2:M(R\tau)}$$

$$k\ \frac{\Sigma(k)=K\tau}{\Gamma\vdash_\Sigma k:K\tau}\qquad \text{callcc}\ \frac{\Gamma,x:K\tau\vdash_\Sigma e:M\tau}{\Gamma\vdash_\Sigma \mathit{callcc}\ x.e:M\tau}$$

$$\text{throw}\ \frac{\Gamma\vdash_\Sigma e_1:K\tau\quad \Gamma\vdash_\Sigma e_2:M\tau}{\Gamma\vdash_\Sigma \mathit{throw}\ e_1\ e_2:M\tau'}$$

Table 3: Type System for $\mathsf{MML}^{SK}_{\mathit{fix}}$

- Types $\boxed{\tau\in\mathsf{T}::=b\mid\tau_1\to\tau_2\mid M\tau\mid R\tau\mid K\tau}$

- Terms
$$\boxed{\begin{array}{lll} e\in\mathsf{E} & ::= & x\mid\lambda x.e\mid e_1e_2\mid \mathit{fix}\ x.e\mid\\ & & \mathit{ret}\ e\mid \mathit{do}\ x\leftarrow e_1;e_2\mid \mathit{Mfix}\ x.e\mid\\ & & l\mid \mathit{new}\ e\mid \mathit{get}\ e\mid \mathit{set}\ e_1\ e_2\mid\\ & & k\mid \mathit{callcc}\ x.e\mid \mathit{throw}\ e_1e_2\end{array}}$$

The type $K\tau$ is the type of continuations which can be invoked on arguments of type $M\tau$ (invoking the continuation aborts the current context). The expression *callcc* $x.e$ binds the current continuation to x; the expression *throw* e_1e_2 has the dual effect of aborting the current continuation and using e_1 instead as the current continuation. This effectively "jumps" to the point where the continuation e_1 was captured by *callcc*.
Table 3 gives the typing rules for deriving judgments of the form $\Gamma\vdash_\Sigma e:\tau\Gamma$ where $\Gamma:\mathsf{X}\overset{fin}{\to}\mathsf{T}$ is a type assignment for variables $x:\tau$ and $\Sigma:\mathsf{L}\cup\mathsf{K}\overset{fin}{\to}\mathsf{T}$ is a signature for locations $l:R\tau$ and continuations $k:K\tau$.

The *simplification* relation $\longrightarrow$ on terms is given by the compatible closure of the following rewrite rules:

β) $(\lambda x.e_2)e_1 \longrightarrow e_2\{x := e_1\}$

fix) $\mathit{fix}\ x.e \longrightarrow e\{x := \mathit{fix}\ x.e\}$

We write $=$ for the equivalence induced by $\longrightarrow$Γi.e. the reflexiveΓsymmetric and transitive closure of $\longrightarrow$. We state the properties of simplification relevant for our purposes.

Proposition 6.1 (Congr) *The equivalence* $=$ *induced by* $\longrightarrow$ *is a congruence.*

Proposition 6.2 (CR) *The simplification relation* $\longrightarrow$ *is confluent.*

Proposition 6.3 (SR) *If* $\Gamma \vdash_\Sigma e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash_\Sigma e' : \tau$.

To define the computation relation $Id \longmapsto Id' \mid \mathsf{done} \mid \mathsf{err}$ (see Table 4)Γ we need the auxiliary notions of evaluation contextsΓstoresΓcontinuation environmentsΓconfigurations $Id \in \mathsf{Conf}$Γand computational redexes:

- Evaluation contexts $\boxed{E \in \mathsf{EC} ::= \square \mid E[\mathit{do}\ x \leftarrow \square; e] \mid E[\mathit{Mfix}\ x.\square]}$ (or $E ::= \square \mid \mathit{do}\ x \leftarrow E; e \mid \mathit{Mfix}\ x.E$)
- Stores $\mu \in \mathsf{S} \triangleq \mathsf{L} \xrightarrow{fin} \mathsf{E}$ and continuation environments $\rho \in \mathsf{KE} \triangleq \mathsf{K} \xrightarrow{fin} \mathsf{EC}$
- Configurations $(X|\mu, \rho, e, E) \in \mathsf{Conf} \triangleq \mathcal{P}_{fin}(\mathsf{X}) \times \mathsf{S} \times \mathsf{KE} \times \mathsf{E} \times \mathsf{EC}$ consist of the current store μ and continuation environment ρΓthe program fragment e under consideration and its evaluation context E. The set X records the recursive variables generated so farΓthus X grows as the computation progresses.
- Computational redexes

$$\boxed{\begin{array}{lll} r \in \mathsf{R} & ::= & \mathit{ret}\ e \mid \mathit{do}\ x \leftarrow e_1; e_2 \mid \mathit{Mfix}\ x.e \mid \\ & & \mathit{new}\ e \mid \mathit{get}\ l \mid \mathit{set}\ l\ e \mid \mathit{callcc}\ x.e \mid \mathit{throw}\ k\ e \end{array}}$$

Remark 6.4 In the absence of *Mfix* $x.e$Γthe hole $\square$ of an evaluation context E is never within the scope of a binder. Therefore one can represent E as a λ-abstraction $\lambda x.E[x]$Γwhere $x \notin \mathrm{FV}(E)$. This is how continuations are modeled in the λ-calculusΓin particular the operation $E[e]$ of replacing the hole in E with a term e becomes simplification of the β-redex $(\lambda x.E[x])\ e$. This representation of continuations is adopted also in the reduction semantics of functional languages with control operators [WF94]. In such reduction semantics there is no need keep a continuation environment ρΓbecause a continuation k with $\rho(k) = E$ is represented by the λ-abstraction $\lambda x.E[x]$. In the presence of *Mfix* $x.e$ (or when modeling partial evaluationΓmulti-stage programmingΓand call-by-need [AF97ΓAMO+95ΓMOW98])Γevaluation may take place within the scope of a binderΓand one can no longer represent an evaluation context with a λ-abstractionΓbecause the operation $E[e]$ may capture free variables in e. In this caseΓcontinuation environments are very convenientΓsince the subtle issues regarding variable capture are confined to the level of configurationsΓand do not percolate in terms and other syntactic categories. ∎

In an evaluation context the hole $\square$ can be within the scope of a binderΓthus an evaluation context E has not only a set of free variablesΓbut also a set of captured variables. MoreoverΓthe definition of $E\{x' := e'\}$ differs from the capture-avoiding substitution $e\{x' := e'\}$ for termsΓbecause captured variables cannot be renamed.

Definition 6.5 *The sets* $\mathrm{CV}(E)$ *and* $\mathrm{FV}(E)$ *of captured and free variables and the substitution* $E\{x' := e'\}$ *are defined by induction on* E*:*

- $\mathrm{CV}(\square) \triangleq \mathrm{FV}(\square) \triangleq \emptyset$ *and* $\square\{x' := e'\} \triangleq \square$
- $\mathrm{CV}(do\ x \leftarrow E; e) \triangleq \mathrm{CV}(E)$, $\mathrm{FV}(do\ x \leftarrow E; e) \triangleq \mathrm{FV}(E) \cup (\mathrm{FV}(e) \setminus \{x\})$ *and* $(do\ x \leftarrow E; e)\{x' := e'\} \triangleq do\ x \leftarrow E\{x' := e'\}; e\{x' := e'\}$ *(the bound variable x can be renamed to be different from x' and from any of the free variables of e').*
- $\mathrm{CV}(\mathit{Mfix}\ x.E) \triangleq \{x\} \cup \mathrm{CV}(E)$, $\mathrm{FV}(\mathit{Mfix}\ x.E) \triangleq \mathrm{FV}(E) \setminus \{x\}$ *and* $(\mathit{Mfix}\ x.E)\{x' := e'\} \triangleq \begin{cases} \mathit{Mfix}\ x.E & \text{if } x = x' \\ \mathit{Mfix}\ x.E\{x' := e'\} & \text{otherwise} \end{cases}$ *(the captured variable x cannot be renamed; free occurrences of x in e' may be captured.)*

The confluent simplification relation $\longrightarrow$ on terms extends in the obvious way to a confluent relation (denoted $\longrightarrow$) on storesΓevaluation contextsΓ computational redexes and configurations.

Lemma 6.6

1. *If* $(X|\mu, \rho, e, E) \longrightarrow (X'|\mu', \rho', e', E')$*, then* $X = X'$, $\mathrm{dom}(\mu') = \mathrm{dom}(\mu)$, $\mathrm{dom}(\rho') = \mathrm{dom}(\rho)$ *and*
    - $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$, $\mathrm{CV}(E') = \mathrm{CV}(E)$ *and* $\mathrm{FV}(E') \subseteq \mathrm{FV}(E)$
    - $\mathrm{FV}(\mu'\ l) \subseteq \mathrm{FV}(\mu\ l)$ *for* $l \in \mathrm{dom}(\mu)$
    - $\mathrm{CV}(\rho'\ k) = \mathrm{CV}(\rho\ k)$ *and* $\mathrm{FV}(\rho'\ k) \subseteq \mathrm{FV}(\rho\ k)$ *for* $k \in \mathrm{dom}(\rho)$

Administrative steps

(A.0) $(X|\mu, \rho, ret\ e, \Box) \longmapsto$ done

(A.1) $(X|\mu, \rho, do\ x \leftarrow e_1; e_2, E) \longmapsto (X|\mu, \rho, e_1, E[do\ x \leftarrow \Box; e_2])$

(A.2) $(X|\mu, \rho, ret\ e_1, E[do\ x \leftarrow \Box; e_2]) \longmapsto (X|\mu, \rho, e_2\{x := e_1\}, E)$

Steps for recursive monadic binding

(M.1) $(X|\mu, \rho, Mfix\ x.e, E) \longmapsto (X, x|\mu, \rho, e, E[Mfix\ x.\Box])$ with x renamed to avoid clashes with X

(M.2) $(X|\mu, \rho, ret\ e, E[Mfix\ x.\Box]) \longmapsto (X|\overline{\mu}, \overline{\rho}, ret\ \overline{e}, \overline{E})$ where $\overline{\bullet}$ stands for $\bullet\{x := fix\ x.ret\ e\}$
(the free occurrences of the recursive variable x are replaced anywhere in the configuration)

(err) $(X|\mu, \rho, x, E) \longmapsto$ err where $x \in X$ (attempt to use an unresolved recursive variable)

Imperative steps

(new) $(X|\mu, \rho, new\ e, E) \longmapsto (X|\mu\{l : e\}, \rho, ret\ l, E)$ where $l \notin \mathrm{dom}(\mu)$

(get) $(X|\mu, \rho, get\ l, E) \longmapsto (X|\mu, \rho, ret\ e, E)$ with $e = \mu(l)$

(set) $(X|\mu, \rho, set\ l\ e, E) \longmapsto (X|\mu\{l = e\}, \rho, ret\ l, E)$ with $l \in \mathrm{dom}(\mu)$

Control steps

(callcc) $(X|\mu, \rho, callcc\ x.e, E) \longmapsto (X|\mu, \rho\{k : E\}, e\{x := k\}, E)$ where $k \notin \mathrm{dom}(\rho)$

(throw) $(X|\mu, \rho, throw\ k\ e, E) \longmapsto (X|\mu, \rho, e, E_k)$ with $E_k = \rho(k)$

Table 4: Computation Relation for MML^{SK}_{fix}

$$(\Box)\ \frac{}{\Delta, \Box\colon M\tau \vdash_\Sigma \Box\colon M\tau}$$

$$(\mathrm{do})\ \frac{\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau_1 \quad \Delta, x\colon \tau_1 \vdash_\Sigma e\colon M\tau_2}{\Delta, \Box\colon M\tau \vdash_\Sigma \mathit{do}\ x \leftarrow E; e\colon M\tau_2}$$

$$(\mathrm{Mfix})\ \frac{\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'}{\Delta, \Box\colon M\tau \vdash_\Sigma \mathit{Mfix}\ x.E\colon M\tau'}\ \Delta(x) = M\tau'$$

Table 5: Well-formed Evaluation Contexts for MML^{SK}_{fix}

2. *If* $(X|\mu, \rho, e, E) \longmapsto (X'|\mu', \rho', e', E')$ *and* $\mathrm{FV}(\mu, \rho, e, E) \cup \mathrm{CV}(\rho, E) \subseteq X$, *then* $X \subseteq X'$, $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$, $\mathsf{dom}(\rho) \subseteq \mathsf{dom}(\rho')$ *and* $\mathrm{FV}(\mu', \rho', e', E') \cup \mathrm{CV}(\rho', E') \subseteq X'$.

Theorem 6.7 (Bisim) *If* $Id \equiv (X|\mu, \rho, e, E)$ *with* $e \in \mathsf{R}$ *and* $Id \xrightarrow{*} Id'$, *then*

1. $Id \longmapsto D$ *implies* $\exists D'$ *s.t.* $Id' \longmapsto D'$ *and* $D \xrightarrow{*} D'$
2. $Id' \longmapsto D'$ *implies* $\exists D$ *s.t.* $Id \longmapsto D$ *and* $D \xrightarrow{*} D'$

where D *and* D' *range over* $\mathsf{Conf} \cup \{\mathsf{done}, \mathsf{err}\}$.

6.1 Type Safety

The definitions of well-formed configurations $\Delta \vdash_\Sigma Id\colon \tau'$ and evaluation contexts $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ must take into account the set X. Thus we need a type assignment Δ mapping $x \in X$ to computational types $M\tau$.

Definition 6.8 $\Delta \vdash_\Sigma (X|\mu, \rho, e, E)\colon \tau' \overset{\Delta}{\iff} \mathsf{dom}(\Sigma) = \mathsf{dom}(\mu) \uplus \mathsf{dom}(\rho)$, $\mathsf{dom}(\Delta) = X$ *and exists* τ *such that*

- $\Delta \vdash_\Sigma e\colon M\tau$ *is derivable*
- $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ *is derivable (see Table 5)*
- $e_l = \mu(l)$ *and* $R\tau_l = \Sigma(l)$ *implies* $\Delta \vdash_\Sigma e_l\colon \tau_l$
- $E_k = \rho(k)$ *and* $K\tau_k = \Sigma(k)$ *implies* $\Delta, \Box\colon M\tau_k \vdash_\Sigma E_k\colon M\tau'$.

The formation rules of Table 5 for deriving $\Delta, \Box\colon M\tau \vdash_\Sigma E\colon M\tau'$ ensure that Δ assigns a computational type to all captured variables of E. We can now formulate the SR and progress properties for MML^{SK}_{fix}.

Theorem 6.9 (SR)

1. *If* $\Delta \vdash_\Sigma Id_1 : \tau'$ *and* $Id_1 \longrightarrow Id_2$*, then* $\Delta \vdash_\Sigma Id_2 : \tau'$

2. *If* $\Delta_1 \vdash_{\Sigma_1} Id_1 : \tau'$ *and* $Id_1 \longmapsto Id_2$*, then exists* $\Sigma_2 \supseteq \Sigma_1$ *and* $\Delta_2 \supseteq \Delta_1$ *s.t.* $\Delta_2 \vdash_{\Sigma_2} Id_2 : \tau'$.

Theorem 6.10 (Progress) *If* $\Delta \vdash_\Sigma (X|\mu, \rho, e, E) : \tau'$*, then one of the following holds*

1. $e \notin \mathsf{R}$ *and* $e \longrightarrow$ *, or*

2. $e \in \mathsf{R}$ *and* $(X|\mu, \rho, e, E) \longmapsto$

6.2 Counter-examples

The left-shrinking property states that:

$$\textit{Mfix } x.(\textit{do } x_1 \leftarrow e_1; e_2) = \textit{do } x_1 \leftarrow e_1; \textit{Mfix } x.e_2 \qquad \text{when } x \notin \mathrm{FV}(e_1)$$

It is instructive to consider how this property fails in MML^{SK}_{fix}. Our example (inspired by examples by Bawden and Carlsson) uses continuations in a way that requires recursive types which can be declared as follows in Haskell syntax:

```
data XT m = XT (m (Int, XT m))  -- final result
data KT m = KT (K (RT m))       -- recursive continuations
data RT m =                     -- arguments to continuations
   Final (XT m)
 | Pair (Bool, KT m)
```

Now we consider the following instance of the left-hand side (again in Haskell syntax):

```
t1 =
 Mfix (\x ->
   do p <- callcc (\k -> return (Pair (True, KT k)))
      case p of
        Pair (b, KT k) ->
          if b
           then
             do Final v <- callcc (\c ->
                              throw k (return (Pair (False, KT c))))
                return (1,v)
           else throw k (return (Final (XT x))))
```

In our semantics (extended with simplification rules for booleansΓ pairsΓ etc) the example evaluates as follows. The pair p initially refers to a continuation which re-binds p. In the then-branch which is initially takenΓ this continuation is invoked with a new pair containing the continuation `c`. This latter continuation expects a value `v` which it includes in the final result `(1,v)`. In the `else`-branch which is taken the second timeΓthat value `v` is bound to `Final (XT x)`. Hence the return value of the body of the *Mfix* is `(1,Final (XT x))` and the entire expression evaluates to `fix x. return (1, Final (XT x))` which is a recursive pair of ones. However were we to move the first *callcc*-expression (which has no free occurrences of `x`) outside the *Mfix*Γthe continuations `k` and `c` would have no access to the variable `x` and the example would evaluate to `return (1,x)` which would cause an error if the second component is needed. The fact that this result is an approximation of the left-hand side does not generalize: with a slightly more complicated exampleΓit is possible to get a different observable value.

Our semantics also differs from the Scheme semantics. The difference in this case is due to the nature of variables in both systems: in our setting variable are bound to expressions and locations must be created and dereferenced explicitly. In Scheme variables implicitly refer to locationsΓwhich means that continuations captured within the body of an *Mfix* not only have access to the free occurrences of the recursive variable in the body of the recursive definition but also to the *location* in which the result is to be stored: this additional expressiveness for continuations invalidates even more transformations like *Mfix* $x.e = e$ when $x \notin FV(e)$ [Baw88]. Such transformations should still be valid in our model.

Acknowledgments We would like to thank Levent Erkök and Magnus Carlsson for very fruitful discussions and comments.

References

[AF97] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*Γ 7(3):265–301Γ May 1997.

[AFMZ02] D. AnconaΓS. FagorziΓE. MoggiΓand E. Zucca. Mixin modules and computational effects. SubmittedΓ2002.

[AMO$^+$95] Zena M. AriolaΓ John MaraistΓ Martin OderskyΓ Matthias FelleisenΓand Philip Wadler. A call-by-need lambda calculus. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*Γpages 233–246ΓNew YorkΓNYΓUSAΓ1995. ACM Press.

[Bar84] H[endrik] P[ieter] Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-HollandΓrevised editionΓ1984.

[Baw88] Alan Bawden. Letrec and callcc implement references. Message to `comp.lang.scheme`Γ1988.

[Bou01] Gérard Boudol. The recursive record semantics of objects revisited. *Lecture Notes in Computer Science*Γ2028:269–283Γ2001.

[Car03] Magnus Carlsson. Value recursion in the continuation monad. Unpublished NoteΓJanuary 2003.

[EL00] Levent Erkök and John Launchbury. Recursive monadic bindings. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*Γvolume 35.9 of *ACM Sigplan Notices*Γpages 174–185ΓN.Y.ΓSeptember 18–21 2000. ACM Press.

[ELM02] Levent ErkökΓJohn LaunchburyΓand Andrew Moran. Semantics of value recursion for monadic input/output. *Journal of Theoretical Informatics and Applications*Γ36(2):155–180Γ2002.

[Erk02] Levent Erkök. *Value Recursion in Monadic Computations.* PhD thesisΓOGI School of Science and EngineeringΓOHSUΓPortlandΓOregonΓ2002.

[FS00] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical Report 546ΓComputer Science DepartmentΓIndiana UniversityΓDecember 2000.

[Jon99] Report on the programming language Haskell 98ΓFebruary 1999.

[KCE98] Richard KelseyΓWilliam ClingerΓand Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*Γ33(9):26–76ΓSeptember 1998.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*Γ6(4):308–320ΓJanuary 1964.

[MF03] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *FoSSaCS 2003*ΓLNCS. Springer-VerlagΓ2003.

[MOW98] John MaraistΓMartin OderskyΓand Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*Γ8(3):275–317ΓMay 1998.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*Γ115(1):38–94Γ1994.

Hierarchies in μ-calculus

Damian Niwiński

Institute of Informatics, Warsaw University

Abstract

Finite–state recognizability is obviously at the basic level of all reasonable complexity hierarchies, as far as computations of finite duration are considered (e.g., over integers). This property is no more true if we deal with infinite computations, running over reals (i.e., infinite words) or, more generally, over infinite trees. In particular, it is well–known that a finite–state Rabin automaton can recognize the set of (suitable encodings of) well-founded trees which is $\mathbf{\Pi}_1^1$ complete in terms of projective hierarchy, and hence not arithmetical, and even not Borel.

An alternation hierarchy of the μ-calculus seems to be well-suited for measuring complexity of infinite computations. It also reconciles the finite–state recognizability and the classical (arithmetical/analytical) hierarchies via some apparent connections, notably between the class $\mu\nu$ (of the alternation hierarchy) and $\mathbf{\Pi}_1^1$. Yet some deeper connections remain to be understood, in particular the refinement of the alternation hierarchy by the Wadge equivalence.

One of the general principles illustrated by both arithmetical and analytical hierarchies is the duality between *separation* and *reduction* property: a class C has separation property iff its dual $\bar{C}$ has reduction property, but a class cannot enjoy both properties. There are some evidences that the principle should also hold for the fixed–point alternation hierarchy, in spite of the limited expressive power of the μ-calculus, not allowing for the diagonal argument.

In the talk I will outline the known connections between various hierarchies and some challenging open problems.

An Alternative Characterization for Complete Iterativeness (Extended Abstract)

Tarmo Uustalu[1] and Varmo Vene[2]

[1] Inst. of Cybernetics, Tallinn Technical University
Akadeemia tee 21, EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

[2] Dept. of Computer Science, University of Tartu
J. Liivi 2, EE-50409 Tartu, Estonia
varmo@cs.ut.ee

Moss [4] and Aczel, Adámek et al. [1] have recently shown that the term algebra of non-wellfounded terms in a universal-algebraic signature gives rise to a monad which is completely iterative in the sense of solvability of arbitrary systems of guarded equations. Aczel, Adámek et al. [2] have moreover shown that it is the free completely iterative monad generated by this signature.

Technically, complete iterativeness is defined for ideal monads as unique existence of an operation on morphisms of a certain type. We show that the concept admits an alternative definition where the criterion is unique existence of a natural transformation, a restriction however being that this definition can only be invoked under the existence of certain final coalgebras. We argue that reasoning about complete iterativeness can sometimes be easier resorting to the alternative definition, one of the reasons being that the diagram chase format is not ideally suited for reasoning about operations on morphisms. The alternative definition is especially useful, if the core of an argument has to be conducted in the category of endofunctors on the base category, as is the case with arguments concerning algebras of terms in binding signatures.

Ideal monads, completely iterative monads The concept of complete iterativeness is defined for monads that are ideal. A monad (T, η, μ) on $\mathcal{C}$ is said to be *ideal*, if it comes together with an endofunctor T' on $\mathcal{C}$ and natural transformations $\tau : T' \to T$, $\mu' : T' \cdot T \to T'$ such that $[\eta, \tau] : \mathsf{Id} + T' \to T$ is a natural isomorphism and

$$\begin{tikzcd}
T' \cdot T \arrow[r, "\tau \cdot T"] \arrow[d, "\mu'"'] & T \cdot T \arrow[d, "\mu"] \\
T' \arrow[r, "\tau"'] & T
\end{tikzcd}$$

An ideal monad $(T, \eta, \mu, T', \tau, \mu')$ is said to be *completely iterative*, if for any guarded equation system with unknowns in A and parameters in B, i.e., a morphism $f : A \to B + T'(A + B)$, there exists a unique morphism $h : A \to TB$

(notation solve(f)) that solves it, i.e., satisfies

$$\begin{tikzcd}
B+T'(A+B) \arrow[d, "\mathsf{inr}_{A,B}+\mathsf{id}_{T'(A+B)}"'] & & A \arrow[ll, "f"'] \arrow[dd, "h"] \\
(A+B)+T'(A+B) \arrow[d, "{[\eta_{A+B},\tau_{A+B}]}"'] & & \\
T(A+B) \arrow[r, "{T[h,\eta_B]}"'] & TTB \arrow[r, "\mu_B"'] & TB
\end{tikzcd}$$

or, which is equivalent (because of the condition relating μ and μ'),

$$\begin{tikzcd}
B+T'(A+B) \arrow[d, "{\mathsf{id}_B+T'[h,\eta_B]}"'] & A \arrow[l, "f"'] \arrow[dd, "h"] \\
B+T'TB \arrow[d, "\mathsf{id}_B+\mu'_B"'] & \\
B+T'B \arrow[r, "{[\eta_B,\tau_B]}"'] & TB
\end{tikzcd} \tag{1}$$

The main result of [2] was that, if an endofunctor H on $\mathcal{C}$ is iteratable (in the sense of existence of the final $(A+H-)$-coalgebra for every $\mathcal{C}$-object A), then the monad structure on the endofunctor T on $\mathcal{C}$ given by $TA = \nu(A+H-)$ is the free completely iterative monad generated by H. In [3], it was shown that iteratability of H is necessary in order that the free H-generated completely iterative monad exists.

An alternative definition Assume that the final $(A+T'(-+A))$-coalgebra exists for every $\mathcal{C}$-object A. Set $(T^\infty A, \omega_A) = (\nu(A+T'(-+A)), \mathsf{out}_{A+T'(-+A)})$. Then one can show that $(T,\eta,\mu,T',\tau,\mu')$ is a completely iterative monad if and only if a unique natural transformation $h : T^\infty \to T$ (notation μ^∞) exists such that

$$\begin{tikzcd}
A+T'(T^\infty A+A) \arrow[d, "{\mathsf{id}_A+T'[h,\eta_A]}"'] & T^\infty A \arrow[l, "\omega_A"'] \arrow[dd, "h"] \\
A+T'TA \arrow[d, "\mathsf{id}_A+\mu'_A"'] & \\
A+T'A \arrow[r, "{[\eta_A,\tau_A]}"'] & TA
\end{tikzcd} \tag{2}$$

The definitions of solve$(-)$ and μ^∞ via each other are: $\mu^\infty_A = \mathsf{solve}(\omega_A)$ and $\mathsf{solve}(f) = \mu^\infty_B \circ \mathsf{Coit}_{B+T'(-+B)}(f)$ $(f : A \to B+T'(A+B))$. By Coit, we denote coiteration: Coit_F takes a F-coalgebra structure map to the corresponding final coalgebra homomorphism.

Notice that morphisms ω_A are guarded equation systems and the condition asserts their unique solvability, so the alternative characterization replaces the requirement of unique solvability of arbitrary guarded equation systems by that of only some specific guarded equation systems which are representative of all others. This makes the relationship between μ^∞ and solve$(-)$ analogous to that between μ and $-^*$ (the Kleisli extension operation). While $-^*$ takes any substitution rule to the corresponding substitution function, μ delivers only those substitution functions that correspond to an identity substitution rule, since $\mu_B = \mathsf{id}_{TB}{}^\star$. Nevertheless μ determines all substitution functions, as $f^\star = \mu_B \circ Tf$ $(f : A \to TB)$.

Intuitively, the decomposition $\mathsf{solve}(f) = \mu_B^\infty \circ \mathsf{Coit}_{B+T'(-+B)}(f)$ refers to solving a guarded equation system with unknowns in A and parameters in B in two stages: first, a "quasi-solution" is calculated which assigns to the elements of A not terms over B (elements of TB), but elements of $T^\infty B$ ("quasi-terms" over B), and subsequently these quasi-terms are "flattened" into terms proper yielding the real solution. (Compare this to calculating the result of substituting a term for all occurences of a certain variable in a term by first naively replacing the variable at these occurrences by the term in question and then flattening the result into a term proper). To provide a contrast, let us note that a non-guarded equation system with unknowns from A and parameters from B is a morphism $f : A \to T(A + B)$ and any such induces a morphism $\mathsf{Coit}_{T(-+B)}(f) : A \to T^\sharp B$ where $T^\sharp B = \nu(T(- + B))$, so non-guarded equation systems with parameters from B are quasi-solvable in terms of elements of $T^\sharp B$. But for T given by $TA = \nu(A + H-)$ (the algebra of non-wellfounded terms over A in signature H) there can be no hope in general to construct a natural transformation $T^\sharp \to T$.

Applications The alternative characterization can be used to prove that the monad structure on $T = \nu(\mathsf{Id} + \mathcal{H}-)$ where $\mathcal{H} : [\mathcal{C},\mathcal{C}] \to [\mathcal{C},\mathcal{C}]$ is given by $\mathcal{H}\mathcal{X} = \mathcal{X}\times\mathcal{X}+\mathcal{X}\cdot(K_1+\mathsf{Id})$ (the algebra of non-wellfounded de Bruijn notations) is completely iterative by explicitly constructing a candidate for μ^∞ and checking that it verifies the required property of being the unique h satisfying (2).

Acknowledgements This research was supported by the Estonian Science Foundation under grant No. 5567. The participation of T. Uustalu and V. Vene at ETAPS 2003 was made possible by a travel grant from the Estonian Information Technology Foundation.

References

1. P. Aczel, J. Adámek, and J. Velebil. A coalgebraic view of infinite trees and iteration. In A. Corradini, M. Lenisa, and U. Montanari, eds., *Proc. of 4th Int. Wksh. on Coalgebraic Methods in Comput. Sci., CMCS'01 (Genova, Apr. 2001)*, vol. 44(1) of *Electr. Notes in Theor. Comput. Sci.*. Elsevier, 2001.
2. P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.*, to appear.
3. S. Milius. On iteratable endofunctors. In *Proc. of 9th Int. Conf. on Category Theory and Comput. Sci., CTCS 2002 (Ottawa, Aug. 2002), Electr. Notes in Theor. Comput. Sci.*, Elsevier, to appear.
4. L. S. Moss. Parametric corecursion. *Theor. Comput. Sci.*, 260(1–2):139–163, 2001.

EASST

EAPLS

EATCS

acm

HIGH-LEVEL SCIENTIFIC CONFERENCES

European Joint Conferences on
THEORY
AND
PRACTICE OF
SOFTWARE

Information Society
Technologies

OFFICE OF NAVAL RESEARCH
IFO
INTERNATIONAL FIELD OFFICE

EUROPEAN OFFICE of AEROSPACE R&D

Microsoft
Research

WARSZAWA

POLSKIE LINIE LOTNICZE
LOT