RWTH Aachen

Department of Computer Science
Technical Report

Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

Klaus Indermark and Thomas Noll

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2004-08

RWTH Aachen · Department of Computer Science · December 2004

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

Klaus Indermark and Thomas Noll

Lehrstuhl für Informatik II
Aachen University of Technology (RWTH)
D–52056 Aachen, Germany
Email: {indermark,noll}@cs.rwth-aachen.de

Abstract. Adding appropriate strictness information to recursive function definitions we achieve a uniform treatment of lazy and eager evaluation strategies. By restriction to first–order functions over basic types we develop a pure stack implementation that avoids a heap even for lazy arguments. We present algebraic definitions of denotational, operational, and stack–machine semantics and prove their equivalence by means of structural induction.

1 Introduction

Recursive definitions play a fundamental rôle in computer science as they offer two different semantic views. From a denotational perspective they can be regarded as equations defining objects as solutions, whereas operationally they may be taken as rewrite rules which produce results by stepwise reduction. The equivalence of these views accounts for the central importance of recursion being at the same time declarative and executable.

This holds in particular for the recursive definition of functions in functional programming languages where a compiler automatically transforms such definitions into executable code. In this paper, we restrict to the special case of first–order function definitions over basic types. For such definitions we develop a stack implementation which does not require any heap nor closures, even for lazy evaluation. We present algebraic correctness proofs for the generation of stack code. Adding appropriate strictness information to a recursive function definition we achieve a uniform treatment of lazy and eager evaluation strategies (cf. [Wad96] for a thorough discussion of these concepts). For that purpose we demand that every function definition indicates for each of its arguments whether it enforces strictness or not [Chi97].

In order to motivate this approach we consider the following simple example with functions on the set $\mathbb{N}$ of non–negative integers:

$$\begin{aligned} F(x) &= G(x-1, H(x)) \\ G(x,y) &= \mathsf{if}\ x=0\ \mathsf{then}\ x\ \mathsf{else}\ G(x-1,y)+y \\ H(x) &= H(x+1) \end{aligned}$$

Here, the second equation gives a primitive recursive definition of the multiplication function, which is called in the first equation with an undefined second argument.

1.1 Operational Semantics

We begin with an operational view and interpret equations as rewrite rules. For a proper implementation, the non–deterministic character of the corresponding rewrite process requires an evaluation order. We present three strategies and show their differences in computing the value of $F(1)$.

1.1.1 Leftmost–Innermost Reduction All function arguments are evaluated before executing a function call; this strategy is also known as **call by value**. It induces for the computation of $F(1)$ the following infinite rewrite process:

$$\begin{aligned} F(1) &\Rightarrow G(1-1, H(1)) \\ &\Rightarrow G(0, H(1)) \\ &\Rightarrow G(0, H(1+1)) \\ &\Rightarrow G(0, H(2)) \\ &\Rightarrow G(0, H(2+1)) \\ &\;\;\vdots \end{aligned}$$

We therefore conclude that $F(1)$ is not defined.

1.1.2 Leftmost–Outermost Reduction This rewriting strategy corresponds to the well–known **call by name** principle where function calls are executed first so that arguments are evaluated only if necessary:

$$\begin{aligned} F(1) &\Rightarrow G(1-1, H(1)) \\ &\Rightarrow \mathsf{if}\ 1-1=0\ \mathsf{then}\ 1-1\ \mathsf{else}\ G((1-1)-1, H(1)) + H(1) \\ &\Rightarrow \mathsf{if}\ 0=0\ \mathsf{then}\ 1-1\ \mathsf{else}\ G((1-1)-1, H(1)) + H(1) \\ &\Rightarrow \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 1-1\ \mathsf{else}\ G((1-1)-1, H(1)) + H(1) \\ &\Rightarrow 1-1 \\ &\Rightarrow 0 \end{aligned}$$

In this case, the computation yields the result $F(1) = 0$.

1.1.3 Mixed Reduction The inefficiency of call by name is obvious: in the above rewriting sequence, the expression $1-1$ is evaluated twice due to multiple occurrences of x in the defining term of G. But as the first argument of G has to be computed anyway, we can do this before calling G, thus avoiding its repeated computation:

$$\begin{aligned} F(1) &\Rightarrow G(1-1, H(1)) \\ &\Rightarrow G(0, H(1)) \\ &\Rightarrow \mathsf{if}\ 0=0\ \mathsf{then}\ 0\ \mathsf{else}\ G(0-1, H(1)) + H(1) \\ &\Rightarrow \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 0\ \mathsf{else}\ G(0-1, H(1)) + H(1) \\ &\Rightarrow 0 \end{aligned}$$

So we realize that this mixture of the first two strategies, delaying the computation of only one function argument, produces shorter computations.

1.2 Denotational Semantics

Now we are going to model these operational differences on a purely denotational level. We therefore have to look for suitable techniques to solve our system of equations, defining the same functions as the previously discussed evaluation strategies. For the second equation

$$G(x, y) = \mathsf{if}\ x = 0\ \mathsf{then}\ x\ \mathsf{else}\ G(x - 1, y) + y$$

there seems to be just one solution, namely the function

$$g : \mathbb{N}^2 \to \mathbb{N} \quad \text{with} \quad g(a, b) = a \cdot b.$$

However, the third equation

$$H(x) = H(x + 1)$$

has many solutions, namely, for each $k \in \mathbb{N}$,

$$h_k : \mathbb{N} \to \mathbb{N} \quad \text{with} \quad h_k(a) = k \quad \text{for every } a \in \mathbb{N}.$$

and, in addition, the partial function

$$h : \mathbb{N} \dashrightarrow \mathbb{N} \quad \text{with} \quad h(a) = \text{undefined} \quad \text{for every } a \in \mathbb{N}.$$

It should be clear that only the partial function h corresponds to our functional view because any argument a yields an infinite computation. Moreover, h turns out to be the least solution with respect to the partial order defined by graph inclusion.

It remains to solve the first equation

$$F(x) = G(x - 1, H(x)).$$

This seems to be a simple task. We just substitute the solutions g and h for the function variables G and H and get

$$f : \mathbb{N} \dashrightarrow \mathbb{N} \quad \text{with} \quad f(a) = g(a - 1, h(a)).$$

And here we have reached the crucial point because we are faced with the problem of how to compose partial functions. Technically, this will be described by introducing a new element $\perp$ that represents undefinedness. A partial function $\varphi : \mathbb{N}^n \dashrightarrow \mathbb{N}$ can then be regarded as a total function $\bar{\varphi} : \mathbb{N}^n \to \mathbb{N}_\perp$ where $\mathbb{N}_\perp := \mathbb{N} \cup \{\perp\}$. For the composition of such functions we have to choose appropriate extensions $\hat{\varphi} : {\mathbb{N}_\perp}^n \to \mathbb{N}_\perp$ which define the behaviour on undefined arguments. We shall see that a recursive function definition allows in a natural way several such extension methods which prove to be an exact denotational counterpart to the operational evaluation strategies.

1.2.1 Strict Extension Let us first assume that an undefined argument always implies an undefined function value. In that case, the argument is not passed to the defining term for evaluation. For our example this means that

$$\begin{aligned} f_s(1) &= g_s(0, h_s(1)) \\ &= g_s(0, \perp) \\ &= \perp \end{aligned}$$

according to the enforced strictness of $g_s : {\mathbb{N}_\perp}^2 \to \mathbb{N}_\perp$. Clearly, the strict extension turns out to be semantically equivalent to call by value.

1.2.2 Non–Strict Extension We may alternatively treat $\perp$ as an ordinary value and pass it to the defining term. Note that this method may give the same strict result as before. However, if not all argument variables occur in the defining term or if the conditional skips an undefined case, the result may be a non–strict function. In that case we obtain

$$\begin{aligned} f_n(1) &= g_n(0, h_n(1)) \\ &= g_n(0, \perp) \\ &= 0 \end{aligned}$$

because the defining term for G yields $g_n(0, \perp) = \mathsf{if}\ 0 = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \ldots = 0$. Obviously, this method denotationally models the computation with call by name.

1.2.3 Mixed Extension Finally, as we observed already with evaluation strategies, we can combine both methods and declare for each function argument whether it enforces strictness or whether it is passed unevaluated to the defining term. Turning back to our example we realize that treating the first argument of G strictly in contrast to the second, we get a perfect match to the mixed reduction strategy.

Conclusion: A recursive function definition uniquely specifies its solution only if we add proper strictness information to each function argument.

In order to simplify the formal treatment we assume that those arguments which are treated strictly precede the others. By suitable reordering of function arguments on left– and right–hand sides of all equations this is easily achieved. Therefore, a function variable with n arguments gets a further index σ between 0 and n to indicate that the first σ arguments are treated strictly, in contrast to the remaining ones. This additional information allows a uniform compilation of recursive function definitions into stack code integrating lazy and eager evaluation strategies.

The remainder of this paper is organized as follows. In Section 2 we present the algebraic and order–theoretic foundations required for the formal treatment of recursive function definitions. These are defined in Section 3 together with their denotational and operational semantics and corresponding equivalence proofs. Then Section 4 introduces a stack interpreter as a first abstract implementation, followed by Section 5 where the interpreter is transformed into a compiler.

2 Mathematical Framework

For proving the correctness of a compiler we have to verify that the translation of syntactic objects preserves their semantics. The proof technique strongly depends on the mathematical framework. Here we choose an algebraic approach where syntactic objects can be viewed as abstract structured entities, independent of their concrete representation, and where their semantics is determined by structural induction, without employing any notion of computation. Together with order–theoretic fixed–point techniques we establish a formal setting that allows a concise and rigorous treatment of recursive function definitions including equivalence proofs of their denotational, reduction, and stack semantics.

2.1 Algebraic Foundations

Our algebraic approach is based on the work of GOGUEN, THATCHER, WAGNER, and WRIGHT who showed in [GTWW77] that programs can be understood as elements of a free term algebra with their semantics definable by homomorphisms.

As a recursive function definition deals with data of at least two sorts, a boolean and some other basic sort, it is convenient to work with sorted sets.

Definition 1: *Let S be a non–empty set whose elements are called **sorts**. A set A together with a mapping $\mathsf{sort} : A \to S$ is called S–**sorted**. For $s \in S$ we denote by $A^s := \mathsf{sort}^{-1}(s)$ the set of all elements of A with sort s. Let $f : A \to B$ be a mapping between S–sorted sets A and B. We say that f **preserves sorts** if $f(A^s) \subseteq B^s$ for each $s \in S$.*

Convention: We only consider sets with a unique sorting and therefore omit the mapping sort. Moreover, we always assume implicitly that mappings between sorted sets preserve sorts.

The syntactic basis of recursive function definitions will be given as a collection of function symbols, called a signature.

Definition 2: *Let S be a set of sorts, $D(S) := S^* \times S$ its derived set of function types and F a $D(S)$–sorted set of **function symbols**. Then we call $\Sigma = \langle S, F\rangle$ a **signature**. The elements of $C := \bigcup_{s\in S} C^s$ where $C^s := F^{(\varepsilon,s)}$ are called **constant symbols**.*

We define the semantics of a signature as an algebraic structure interpreting sorts as sets and function symbols as functions on these sets according to their type information.

Definition 3: *Let A be an S–sorted set and $\tau = (w, s) \in D(S)$. We generalize the denotation A^s to A^w using the following cartesian products:*

$$\begin{aligned} A^{\varepsilon} &:= \{()\} \quad \textit{and} \\ A^{s_1 \ldots s_n} &:= A^{s_1} \times \ldots \times A^{s_n} \quad \textit{for } s_1, \ldots, s_n \in S \textit{ and } n > 0. \end{aligned}$$

*Then we call a mapping $f : A^w \to A^s$ a **function** on A of type τ. If in addition $w = \varepsilon$, then f is called a **constant** of type s. Their collections are denoted by*

$$\begin{aligned} \mathbf{F}^{\tau}(A) &:= \{f \mid f : A^w \to A^s\} \quad \textit{and} \\ \mathbf{F}(A) &:= \bigcup_{\tau\in D(S)} \mathbf{F}^{\tau}(A). \end{aligned}$$

Definition 4: *Let $\Sigma = \langle S, F\rangle$ be a signature. A Σ–**algebra** $\mathfrak{A} = \langle A; \alpha\rangle$ consists of an S–sorted set A, the **carrier** of $\mathfrak{A}$, and a mapping $\alpha : F \to \mathbf{F}(A)$. Each $\alpha(f)$ is called a **base function**, and is also denoted by $f_{\mathfrak{A}}$.*

Observe that both F and $\mathbf{F}(A)$ are $D(S)$–sorted such that according to our sort–preserving convention we have $\alpha(F^{\tau}) \subseteq \mathbf{F}^{\tau}(A)$ for every $\tau \in D(S)$.

Given a signature we can construct new syntactic objects as terms of function symbols and variables. Semantically, this corresponds to the derivation of new functions from base functions by composition. We regard terms as elements of a free algebra such that their semantics can be described by homomorphisms.

Definition 5: *Let $\Sigma = \langle S, F\rangle$ be a signature and X an S–sorted set of variables. The Σ–**term algebra** over X*

$$\mathfrak{T}_\Sigma(X) = \langle T_\Sigma(X); \alpha_T\rangle$$

is defined as follows:

- *$T_\Sigma(X)$ is the smallest S–sorted set which contains all variables and which is closed under free application of function symbols, i.e.,*
  - *$X \subseteq T_\Sigma(X)$ and*
  - *for all $(w, s) \in D(S)$ and $f \in F^{(w,s)}$,*
    $(t_1, \ldots, t_n) \in T_\Sigma(X)^w$ implies $ft_1 \ldots t_n \in T_\Sigma(X)^s$.
- *α_T associates with each $f \in F^{(w,s)}$ the following function on terms:*

$$\alpha_T(f) : T_\Sigma(X)^w \to T_\Sigma(X)^s \quad \text{where} \quad \alpha_T(f)(t_1, \ldots, t_n) := ft_1 \ldots t_n.$$

This definition includes for $w = \varepsilon$ the special case of constant symbols. For each $c \in C$ it follows that $\alpha_T(c)() = c$, and therefore $c \in T_\Sigma(X)$.

A term algebra represents a particular algebraic structure in a concrete way using symbol strings in prefix notation. To formalize the underlying structural properties we introduce homomorphisms. The latter are structure–preserving mappings between algebras which play a central rôle in our algebraic treatment of syntax and semantics.

Definition 6: *Let $\mathfrak{A} = \langle A; \alpha\rangle$ and $\mathfrak{B} = \langle B; \beta\rangle$ be Σ–algebras. A mapping $h : A \to B$ is called a **homomorphism**, and is denoted by $h : \mathfrak{A} \to \mathfrak{B}$, if*

$$h(f_\mathfrak{A}(a^w)) = f_\mathfrak{B}(h(a^w))$$

for each $(w, s) \in D(S)$, $f \in F^{(w,s)}$, and $a^w \in A^w$.
(Note that $h : A \to B$ canonically extends to $h : A^w \to B^w$.)

It is easily verified that identities are homomorphisms, and so are compositions of homomorphisms.

The fundamental property of term algebras consists in their free generation. This means that every element can be obtained from a generating subset by application of base functions in a unique way. The following equivalent definition offers greater flexibility in proofs.

Definition 7: *A Σ–algebra $\mathfrak{A} = \langle A; \alpha\rangle$ is said to be **freely generated** by a set $X \subseteq A$ if each assignment $\chi : X \to B$ into an arbitrary Σ–algebra $\mathfrak{B} = \langle B; \beta\rangle$ uniquely extends to a homomorphism*

$$\overline{\chi} : \mathfrak{A} \to \mathfrak{B}.$$

Term algebras confirm the existence of free algebras:

Theorem 8: *The Σ–term algebra $\mathfrak{T}_\Sigma(X)$ is freely generated by X.*

The proof exploits the fact that each term has a unique decomposition into subterms and is omitted. Instead, we prove the following result which captures the essence of abstract syntax, stating that all free algebras over a given signature have the same structure.

Theorem 9: *Let $\mathfrak{A} = \langle A; \alpha \rangle$ and $\mathfrak{B} = \langle B; \beta \rangle$ be Σ–algebras which are freely generated by X. Then $\mathfrak{A}$ and $\mathfrak{B}$ are isomorphic, i.e., there exists a bijective homomorphism $h : \mathfrak{A} \to \mathfrak{B}$.*

Proof: Since X is a subset of both A and B, the identity on X, $\mathsf{id}_X : X \to X$, can be regarded as an inclusion $\mathsf{in}_{X,A} : X \to A$ and also as $\mathsf{in}_{X,B} : X \to B$. By definition, these assignments uniquely extend to homomorphisms $\overline{\mathsf{in}_{X,A}} : \mathfrak{B} \to \mathfrak{A}$ and $\overline{\mathsf{in}_{X,B}} : \mathfrak{A} \to \mathfrak{B}$. Their composition $\overline{\mathsf{in}_{X,A}} \circ \overline{\mathsf{in}_{X,B}}$ is a homomorphism on $\mathfrak{A}$ that coincides with the identical homomorphism on $\mathfrak{A}$ because both extend $\mathsf{in}_{X,A}$. Interchanging the rôles of A and B we conclude that $\overline{\mathsf{in}_{X,B}} \circ \overline{\mathsf{in}_{X,A}} : \mathfrak{B} \to \mathfrak{B}$ is the identical homomorphism on $\mathfrak{B}$. Therefore, $\overline{\mathsf{in}_{X,B}}$ must be injective and surjective, thus satisfying the assertion. $\square$

Abstract syntax: As the semantics of a programming language, and similarly its translation into machine code, depends only on its structural properties, it is possible to regard programs syntactically as elements of a free algebra. Thereby, we abstract the relevant structural information of programs with the advantage of choosing deliberately between suitable concrete representations, due to their isomorphic nature.

The definition of free algebras implies that we can prove properties by means of **structural induction**: for $M \subseteq T_\Sigma(X)$ it holds that $M = T_\Sigma(X)$ iff $X \subseteq M$ and M is closed under all base functions $\alpha_T(f)$ with $f \in F$.

Algebraic semantics: The unique homomorphic extension $\overline{\chi} : \mathfrak{T}_\Sigma(X) \to \mathfrak{A}$ of an assignment $\chi : X \to A$ permits a very simple definition of semantics and similarly of code generation. This extension is also said to be defined by structural induction because $\overline{\chi}$ can be viewed as the unique solution of the following system of equations:

$$\begin{aligned} \overline{\chi}(x) &= \chi(x) \quad \text{for every } x \in X \\ \overline{\chi}(ft_1 \dots t_n) &= f_{\mathfrak{A}}(\overline{\chi}(t_1), \dots, \overline{\chi}(t_n)) \quad \text{for every } ft_1 \dots t_n \in T_\Sigma(X) \end{aligned}$$

Definition 10: *Let $t \in T_\Sigma(X)$, and let $\chi : X \to A$ be an assignment into a Σ–algebra $\mathfrak{A} = \langle A; \alpha \rangle$. Then*

$$[\![t]\!]_{(\mathfrak{A},\chi)} := \overline{\chi}(t) \in A$$

is called the ***algebraic semantics*** *of t with respect to $\mathfrak{A}$ and χ.*

Note that this method of defining semantics is purely denotational and does not require any notion of computation.

There are two special cases for the algebraic semantics $[\![t]\!]_{(\mathfrak{A},\chi)}$: χ may be an inclusion that leaves its arguments unchanged or, conversely, $\mathfrak{A}$ may also be a term algebra so that function symbols are preserved. The corresponding homomorphisms are called evaluation and substitution, respectively.

- **Evaluation:** if $X \subseteq A$ and $\chi = \mathsf{in}_{X,A}$, we simply write $[\![t]\!]_{\mathfrak{A}}$ instead of $[\![t]\!]_{(\mathfrak{A},\chi)}$. For $X = \emptyset$, $[\![t]\!]_{\mathfrak{A}}$ is known as **initial algebra semantics**.

– **Substitution:** If $\chi : X \to T_\Sigma(Y)$, we write its application as usual in postfix notation, $t\chi$, rather than $[\![t]\!]_{(\mathfrak{A},\chi)}$ or $\overline{\chi}(t)$.

We see that any induced homomorphism $\overline{\chi} : T_\Sigma(X) \to \mathfrak{A}$ splits into a substitution $\overline{\mathsf{sub}} : \mathfrak{T}_\Sigma(X) \to \mathfrak{T}_\Sigma(A)$, where $\mathsf{sub}(x) := \chi(x)$ for every $x \in X$, followed by an evaluation $\overline{\mathsf{id}_A} : \mathfrak{T}_\Sigma(A) \to \mathfrak{A}$:

$$\overline{\chi} = \overline{\mathsf{id}_A} \circ \overline{\mathsf{sub}},$$

since both sides coincide on X.

The following lemma describes a slightly more general result of composing an arbitrary substitution with an evaluation. It will be needed later in the order–theoretic context where B is the flat extension of A.

Lemma 11 (Substitution Lemma): *For any substitution* $\overline{\mathsf{sub}} : \mathfrak{T}_\Sigma(X) \to \mathfrak{T}_\Sigma(A)$ *and any evaluation* $\overline{\mathsf{in}_{A,B}} : \mathfrak{T}_\Sigma(A) \to \mathfrak{B}$ *it holds that*

$$\overline{\overline{\mathsf{in}_{A,B}} \circ \mathsf{sub}} = \overline{\mathsf{in}_{A,B}} \circ \overline{\mathsf{sub}}.$$

Proof: Both homomorphisms coincide on X and therefore must be equal:

$$\begin{aligned} \overline{\overline{\mathsf{in}_{A,B}} \circ \mathsf{sub}}(x) &= (\overline{\mathsf{in}_{A,B}} \circ \mathsf{sub})(x) = \overline{\mathsf{in}_{A,B}}(\mathsf{sub}(x)) \quad \text{and} \\ (\overline{\mathsf{in}_{A,B}} \circ \overline{\mathsf{sub}})(x) &= \ \overline{\mathsf{in}_{A,B}}(\overline{\mathsf{sub}}(x)) \ \ = \overline{\mathsf{in}_{A,B}}(\mathsf{sub}(x)). \end{aligned}$$ □

Hence, substitution and evaluation are in a certain way interchangeable. And it is this property that yields the equivalence of denotational and operational semantics.

Term functions: Terms will be used as right–hand sides of equations in order to define new functions. For that purpose we fix an S–sorted **standard alphabet of argument variables**

$$\mathbb{X} := \{x_i^s \mid s \in S, i \in \mathbb{N}\}.$$

We restrict to the definition of proper functions having at least one argument because the recursive definition of constants in flat domains is of little interest. Therefore, we use the notation

$$F(S) := S^+ \times S$$

for proper function types instead of $D(S) = S^* \times S$. Each $w = s_1 \ldots s_n \in S^+$ determines the subset

$$\mathbb{X}_w := \{x_1^{s_1}, \ldots, x_n^{s_n}\}$$

which should not be confused with $\mathbb{X}^w = \mathbb{X}^{s_1} \times \ldots \times \mathbb{X}^{s_n}$. With $x^w := (x_1^{s_1}, \ldots, x_n^{s_n})$ as a non–empty list of argument variables, we abstract from $t \in T_\Sigma(\mathbb{X}_w)^s$ the **explicit function definition**

$$\lambda x^w.t$$

which yields, when interpreted by a Σ–algebra $\mathfrak{A}$, the **term function**

$$[\![\lambda x^w.t]\!]_{\mathfrak{A}} : A^w \to A^s$$

defined as follows. Each argument vector $a^w = (a_1, \ldots, a_n) \in A^w$ determines the assignment $[x^w/a^w] : \mathbb{X}_w \to A$ by $[x^w/a^w](x_i^{s_i}) := a_i$ for $i = 1, \ldots, n$, so that we can define

$$[\![\lambda x^w.t]\!]_{\mathfrak{A}}(a^w) := [\![t]\!]_{(\mathfrak{A},[x^w/a^w])}.$$

In general, we drop the index w and simply write $\bar{x}$ and $\bar{a}$.

Term functions will be employed for giving denotational semantics to recursive function definitions.

In the special case of finite substitutions, as induced by term functions, the Substitution Lemma implies the following result.

Corollary 12: *For* $\overline{[\bar{x}/\bar{u}]} : \mathfrak{T}_\Sigma(\mathbb{X}_w) \to \mathfrak{T}_\Sigma(A)$*,* $\overline{\mathsf{in}_{A,B}} : \mathfrak{T}_\Sigma(A) \to \mathfrak{B}$*, and* $t \in T_\Sigma(\mathbb{X}_w)$*, it holds that*

$$[\![t[\bar{x}/\bar{u}]]\!]_{\mathfrak{B}} = [\![t]\!]_{(\mathfrak{B},[\bar{x}/[\![\bar{u}]\!]_{\mathfrak{B}}])}.$$

Note that according to our extension of sort–preserving mappings, we have for $\bar{u} = (u_1, \ldots, u_n)$ that $[\![\bar{u}]\!]_{\mathfrak{B}} = ([\![u_1]\!]_{\mathfrak{B}}, \ldots, [\![u_n]\!]_{\mathfrak{B}})$ because $[\![u_i]\!]_{\mathfrak{B}} = \overline{\mathsf{in}_{A,B}}(u_i)$.

We shall see that this commutativity between substitution and evaluation provides the essential link between fixed–point and reduction semantics.

2.2 Order–Theoretic Foundations

The explicit definition of term functions did not require any notion of computation. Instead, we used homomorphisms as an algebraic tool to describe the semantics by structural induction. As we have seen in the introduction, a denotational approach to recursive function definitions benefits from the additional use of order–theoretic methods. They allow to define semantics using least fixed points of continuous functions on complete partial orders. Technically, we replace partial functions by continuous functions on flat domains with a new element $\bot$ that represents an undefined value. This also enables us to model strictness properties as a denotational analogue of various implementation strategies.

Definition 13: *Let A be a non–empty set and $\leq \subseteq A \times A$ a binary relation being*

- *reflexive: $a \leq a$,*
- *transitive: $a \leq b$ and $b \leq c$ implies $a \leq c$, and*
- *antisymmetric: $a \leq b$ and $b \leq a$ implies $a = b$ for every $a, b, c \in A$.*

Then $\mathfrak{A} = \langle A; \leq \rangle$ is called a ***partial order****.*

The simplest partial orders occurring in our denotational treatment of recursive function definitions are flat extensions of S–sorted sets. Since undefined values should also have sorts we use the set $\{\bot^s \mid s \in S\}$ for that purpose.

Definition 14: *The* ***flat extension*** *of an S–sorted set A is defined by*

$$\langle A_\bot; \leq \rangle$$

where $A_\bot := \bigcup_{s \in S} A_\bot^s$ and $A_\bot^s := A^s \cup \{\bot^s\}$, and where $a \leq b$ if $\{a, b\} \subseteq A_\bot^s$ and ($a = \bot^s$ or $a = b$) for some $s \in S$.

Obviously, $A_\perp$ is S–sorted and partially ordered by $\leq$. In addition, all $\langle A^s_\perp; \leq \rangle$ with $\leq$ restricted appropriately are partial orders.

Further partial orders are generated by means of product and function spaces which inherit the ordering relations from their components.

Lemma 15: *For partial orders $\mathfrak{A}_1 = \langle A_1; \leq_1 \rangle$ and $\mathfrak{A}_2 = \langle A_2; \leq_2 \rangle$,*

*(i) the **product space** $\mathfrak{A}_1 \times \mathfrak{A}_2 := \langle A_1 \times A_2; \leq \rangle$ where $(a_1, a_2) \leq (b_1, b_2)$ if $a_1 \leq_1 b_1$ and $a_2 \leq_2 b_2$, and*
*(ii) the **function space** $[\mathfrak{A}_1 \to \mathfrak{A}_2] := \langle \{f \mid f : A_1 \to A_2\}; \leq \rangle$ where $f \leq g$ if $f(a) \leq_2 g(a)$ for every $a \in A_1$*

are again partial orders.

It follows that the flat partial orders $\langle A^s_\perp; \leq \rangle$ induce for $w = s_1 \ldots s_n \in S^+$ and $s \in S$

(i) the product space $A^w_\perp := A^{s_1}_\perp \times \ldots \times A^{s_n}_\perp$ and
(ii) the function space $\mathbf{F}^{(w,s)}(A_\perp) := \{f \mid f : A^w_\perp \to A^s_\perp\}$,

being partially ordered as defined above.

Since functions on $A_\perp$ will be our principal semantic objects, we introduce some of their properties. For our purposes, it is the behaviour on $\perp$–arguments that has to be modelled carefully.

Definition 16: *Let $f \in \mathbf{F}^{(w,s)}(A_\perp)$, $w = s_1 \ldots s_n \in S^+$, and $1 \leq i \leq n$.*

*(i) f is called i–**strict** if for each $\bar{a} = (a_1, \ldots, a_n) \in A^w_\perp$ we have $f(\bar{a}) = \perp^s$ whenever $a_i = \perp^{s_i}$, and*
*(ii) f is called **strict** if f is i–strict for every $i = 1, \ldots, n$.*

Note that $f \in \mathbf{F}^{(w,s)}(A)$ extends uniquely to a strict $f_\perp \in \mathbf{F}^{(w,s)}(A_\perp)$. But we cannot confine ourselves to strict functions. The very conditional, indispensable for recursive function definitions, may skip an undefined alternative and yet produce a defined value. Although this behaviour contradicts strictness, it exhibits a more general property insofar as it preserves the partial order that results from the degree of definedness. This means that if the amount of information about an argument increases, this must also hold for the resulting value. Certainly, any computable function has to share this monotonicity.

Definition 17: *Let $\langle A_1; \leq_1 \rangle$ and $\langle A_2; \leq_2 \rangle$ be partial orders and $f : A_1 \to A_2$. f is called **monotonic** if $a \leq_1 b$ implies $f(a) \leq_2 f(b)$ for all $a, b \in A_1$.*

It follows directly that strict functions are monotonic, and that monotonic functions are closed under composition. As we shall deal with monotonic functions only, we fix for each $(w, s) \in F(S)$ the function space

$$\mathbf{mF}^{(w,s)}(A_\perp) := \{f \mid f : A^w_\perp \to A^s_\perp, f \text{ monotonic}\}.$$

Being a subspace of $\mathbf{F}^{(w,s)}(A_\perp)$, it inherits its partial ordering.

The collection of monotonic functions

$$\mathbf{mF}(A_\perp) := \bigcup_{(w,s) \in F(S)} \mathbf{mF}^{(w,s)}(A_\perp)$$

is $F(S)$–sorted so that we can use again the shorthand for cartesian products:

$$\mathbf{mF}^{\tau_1\cdots\tau_r}(A_\perp) := \mathbf{mF}^{\tau_1}(A_\perp) \times \ldots \times \mathbf{mF}^{\tau_r}(A_\perp) \text{ for } \tau_1 \ldots \tau_r \in F(S)^+$$

being a suitable domain for the solution of a recursive function definition.

The flat extension of S–sorted sets not only allows to handle partial as total functions. We also have to explain their behaviour on undefined arguments. This is formally achieved by extending functions on A to monotonic functions on $A_\perp$. From our introductory discussion we know that there exist several possibilities for such extensions which correspond to various implementation strategies.

For simplicity we assume that all base functions except conditionals are extended strictly. From now on, we consider only signatures which include a boolean sort and conditional symbols, and interpret them accordingly.

Definition 18: *A signature $\Sigma = \langle S, F\rangle$ is called* ***branching*** *if S has a special sort* bool *and, for each $s \in S$, there is a conditional symbol* $\mathsf{cond}_s \in F^{(\mathsf{bool}\, ss, s)}$. *Their collection is denoted by* $F_{\mathsf{cond}} := \{\mathsf{cond}_s \mid s \in S\}$, *and we let* $F_{\mathsf{base}} := F \setminus F_{\mathsf{cond}}$ *denote the set of proper base functions.*

A Σ–algebra $\mathfrak{A} = \langle A; \alpha\rangle$ is called ***branching*** *if Σ is branching, and if in addition* bool *and* cond_s *are interpreted as*

- $A^{\mathsf{bool}} = \mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ *and*
- $\alpha(\mathsf{cond}_s) : \mathbb{B} \times A^s \times A^s \to A^s$ *with* $(\mathsf{true}, a, b) \mapsto a$ *and* $(\mathsf{false}, a, b) \mapsto b$ *for each* $s \in S$.

The ***strict extension*** $\mathfrak{A}_\perp = \langle A_\perp; \alpha_\perp\rangle$ *of a branching Σ–algebra $\mathfrak{A} = \langle A; \alpha\rangle$ is defined by extending base functions as follows:*

- $\alpha_\perp(\mathsf{cond}_s)(\perp^{\mathsf{bool}}, a, b) := \perp^s$,
  $\alpha_\perp(\mathsf{cond}_s)(\mathsf{true}, a, b) := a$ *(even if $b = \perp^s$),*
  $\alpha_\perp(\mathsf{cond}_s)(\mathsf{false}, a, b) := b$ *(even if $a = \perp^s$), and*
- $\alpha_\perp(f) := \alpha(f)_\perp$ *for every* $f \in F_{\mathsf{base}} \setminus C$ *and*
  $\alpha_\perp(c) := \alpha(c)$ *for every* $c \in C$.

More generally, a ***monotonic*** *Σ–algebra* $\mathfrak{A}_m = \langle A_\perp; \alpha_m\rangle$ *is defined in the same way as* $\mathfrak{A}_\perp$ *only that we allow arbitrary monotonic extensions for all* $f \in F_{\mathsf{base}} \setminus C$.

We introduced the more general concept of monotonic algebras for technical reasons: in order to define the semantics of recursive function definitions we enlarge the signature Σ by function variables which take arbitrary monotonic functions as values.

Extended term functions: After extending base functions we now turn to term functions. Their extension depends on how we deal with $\perp$–arguments. Either they directly enforce a $\perp$–result, or they affect the term semantics just as the other arguments do. Note that this difference concerns the defining method and not necessarily the extension itself. We shall see below that both methods may produce equal or different results. But first let us refine our treatment of $\perp$–arguments. Each argument of an explicit function definition will get a *strictness tag* that decides whether we skip term evaluation enforcing strictness or not. For simplicity we assume that arguments enforcing strictness precede arguments for term evaluation so that the required strictness information can be given by an index $\sigma \in \{0, \ldots, n\}$ where n is the number of arguments.

Definition 19: *Let Σ be a branching signature, $(w,s) \in F(S)$, $0 \le \sigma \le |w|$, and $t \in T_\Sigma(\mathbb{X}_w)^s$. Then we call*

$$\lambda^\sigma x^w.t$$

*an explicit function definition **with strictness information**. When interpreted by a monotonic Σ–algebra $\mathfrak{A}$, it determines an **extended term function** on $A_\perp$,*

$$[\![\lambda^\sigma x^w.t]\!]_{\mathfrak{A}} : A_\perp^w \to A_\perp^s,$$

where, for $a^w = (a_1, \ldots, a_n) \in A_\perp^w$,

$$[\![\lambda^\sigma x^w.t]\!]_{\mathfrak{A}}(a^w) := \begin{cases} [\![t]\!]_{(\mathfrak{A},[x^w/a^w])} & \text{if } (a_1,\ldots,a_\sigma) \in A^\sigma \\ \perp^s & \text{otherwise} \end{cases}$$

Lemma 20: *Extended term functions are monotonic:*

$$[\![\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}} \in \mathbf{mF}^{(w,s)}(A_\perp).$$

Moreover,

$$[\![\lambda^{|w|}\bar{x}.t]\!]_{\mathfrak{A}} \le [\![\lambda^{|w|-1}\bar{x}.t]\!]_{\mathfrak{A}} \le \ldots \le [\![\lambda^{0}\bar{x}.t]\!]_{\mathfrak{A}},$$

and for $t = f\bar{x}$ with strict $f_{\mathfrak{A}}$, it holds that

$$[\![\lambda^{|w|}\bar{x}.t]\!]_{\mathfrak{A}} = [\![\lambda^{|w|-1}\bar{x}.t]\!]_{\mathfrak{A}} = \ldots = [\![\lambda^{0}\bar{x}.t]\!]_{\mathfrak{A}},$$

whereas for $t = c \in C$ we have

$$[\![\lambda^{|w|}\bar{x}.t]\!]_{\mathfrak{A}} < [\![\lambda^{|w|-1}\bar{x}.t]\!]_{\mathfrak{A}} < \ldots < [\![\lambda^{0}\bar{x}.t]\!]_{\mathfrak{A}}.$$

Proof: First, we prove by induction on $t \in T_\Sigma(\mathbb{X}_w)^s$ that each term function without strictness information, $[\![\lambda\bar{x}.t]\!]_{\mathfrak{A}} : A_\perp^w \to A_\perp^s$, is monotonic.

(i) $t \in \mathbb{X}_w$ yields a projection. Its monotonicity follows from the partial ordering of $A_\perp^w$.
(ii) $t \in C$ yields a constant function, which is clearly monotonic.
(iii) If $t = ft_1 \ldots t_n$ and, by induction hypothesis, all $[\![\lambda\bar{x}.t_i]\!]_{\mathfrak{A}}$ are monotonic, it follows for $\bar{a} \le \bar{b}$ in $A_\perp^w$ that

$$\begin{aligned} [\![\lambda\bar{x}.t]\!]_{\mathfrak{A}}(\bar{a}) &= [\![ft_1 \ldots t_n]\!]_{(\mathfrak{A},[\bar{x}/\bar{a}])} \\ &= f_{\mathfrak{A}}([\![t_1]\!]_{(\mathfrak{A},[\bar{x}/\bar{a}])}, \ldots, [\![t_n]\!]_{(\mathfrak{A},[\bar{x}/\bar{a}])}) \\ &\le f_{\mathfrak{A}}([\![t_1]\!]_{(\mathfrak{A},[\bar{x}/\bar{b}])}, \ldots, [\![t_n]\!]_{(\mathfrak{A},[\bar{x}/\bar{b}])}) \\ &= [\![\lambda\bar{x}.t]\!]_{\mathfrak{A}}(\bar{b}) \end{aligned}$$

by monotonicity of $f_{\mathfrak{A}}$ and of the occurring term functions. Hence, $[\![\lambda\bar{x}.t]\!]_{\mathfrak{A}}$ is monotonic, too.

Now, let $0 \le \sigma \le |w|$ and $\bar{a} \le \bar{b}$ in $A_\perp^w$. If $(a_1,\ldots,a_\sigma) \in A^\sigma$, we also have $(b_1,\ldots,b_\sigma) \in A^\sigma$, so that

$$[\![\lambda^\sigma\bar{x}.t]\!]_{\mathfrak{A}}(\bar{a}) = [\![t]\!]_{(\mathfrak{A},[\bar{x}/\bar{a}])} \le [\![t]\!]_{(\mathfrak{A},[\bar{x}/\bar{b}])} = [\![\lambda^\sigma\bar{x}.t]\!]_{\mathfrak{A}}(\bar{b}).$$

In the other case that $(a_1,\ldots,a_\sigma) \notin A^\sigma$, we get directly

$$[\![\lambda^\sigma\bar{x}.t]\!]_{\mathfrak{A}}(\bar{a}) = \perp^s \le [\![\lambda^\sigma\bar{x}.t]\!]_{\mathfrak{A}}(\bar{b}),$$

and thereby the monotonicity of $[\![\lambda^\sigma\bar{x}.t]\!]_{\mathfrak{A}}$.

The remaining assertions are obvious. □

Term Functionals: Recursive function definitions are built up from terms that also contain function variables besides function symbols. By further abstraction we associate functionals with such terms. These are mappings between function spaces forming the basis of fixed–point semantics. Formally, we choose an $F(S)$–sorted **standard alphabet of function variables**

$$\mathbb{F} := \{F_j^\tau \mid \tau \in F(S), j \in \mathbb{N}\}.$$

Using a similar notation as with argument variables we associate with every $\rho = \tau_1 \dots \tau_r \in F(S)^+$ the set

$$\mathbb{F}_\rho := \{F_1^{\tau_1}, \dots, F_r^{\tau_r}\}$$

which represents the function variables of a recursive function definition. In order to describe its right–hand sides, we enlarge the set F of function symbols by $\mathbb{F}_\rho$ and thus obtain the extended signature

$$\Sigma[\mathbb{F}_\rho] := \langle S, F \cup \mathbb{F}_\rho \rangle.$$

A right–hand side will then be a term $t \in T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X}_w)^s$ where $(w,s) \in F(S)$. It determines for $\bar{F} = (F_1^{\tau_1}, \dots, F_r^{\tau_r})$ and $0 \leq \sigma \leq |w|$ the functional definition

$$\lambda \bar{F}.\lambda^\sigma \bar{x}.t$$

which yields, when interpreted by $\mathfrak{A}_\perp$, the **term functional**

$$[\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp} : \mathbf{mF}^\rho(A_\perp) \to \mathbf{mF}^{(w,s)}(A_\perp),$$

defined as follows. For an argument vector $\bar{g} = (g_1, \dots, g_r) \in \mathbf{mF}^\rho(A_\perp)$, we extend $\mathfrak{A}_\perp$ to the monotonic $\Sigma[\mathbb{F}_\rho]$–algebra $\mathfrak{A}_\perp[\bar{g}] := \langle A_\perp; \alpha_{\bar{g}} \rangle$ with $\alpha_{\bar{g}}(F_j^{\tau_j}) := g_j$ for $j = 1, \dots, r$. This suggests to set

$$[\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp}(\bar{g}) := [\![\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp[\bar{g}]}$$

in analogy to the definition of term functions. It follows from the proof of the previous lemma that the result is in fact a monotonic function since all g_j are. In addition we can prove that the functional itself is monotonic.

Lemma 21: *Each term functional $[\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp}$ is monotonic.*

Proof: We have to check that $[\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp}(\bar{g}) \leq [\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp}(\bar{h})$ whenever $\bar{g} \leq \bar{h}$. By Lemma 15, this holds if $[\![\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp[\bar{g}]}(\bar{a}) \leq [\![\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp[\bar{h}]}(\bar{a})$ for all $\bar{a} \in A_\perp^w$. In case that $(a_1, \dots, a_\sigma) \notin A^\sigma$, this is obvious because we obtain $\perp$ on both sides. Therefore, it suffices to verify that

$$(*) \qquad [\![t]\!]_{(\mathfrak{A}_\perp[\bar{g}],[\bar{x}/\bar{a}])} \leq [\![t]\!]_{(\mathfrak{A}_\perp[\bar{h}],[\bar{x}/\bar{a}])}$$

holds for all $t \in T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X}_w)$ and $\bar{a} \in A_\perp^w$. We prove $(*)$ by induction on t, using the abbreviation

$$[\![t]\!]_{(\bar{g},\bar{a})} := [\![t]\!]_{(\mathfrak{A}_\perp[\bar{g}],[\bar{x}/\bar{a}])}.$$

(i) For $t \in \mathbb{X}_w \cup C$, $(*)$ holds with equality because t does not contain any function variable.

(ii) Let $t = ft_1 \dots t_m$, $f \in F$, and, by induction hypothesis, all t_i satisfy $(*)$. We conclude:

$$\begin{aligned}
[\![ft_1 \dots t_m]\!]_{(\bar{g},\bar{a})} &= f_{\mathfrak{A}_\perp}([\![t_1]\!]_{(\bar{g},\bar{a})}, \dots, [\![t_m]\!]_{(\bar{g},\bar{a})}) \\
&\leq f_{\mathfrak{A}_\perp}([\![t_1]\!]_{(\bar{h},\bar{a})}, \dots, [\![t_m]\!]_{(\bar{h},\bar{a})}) \\
&= [\![ft_1 \dots t_m]\!]_{(\bar{h},\bar{a})}.
\end{aligned}$$

(iii) Let $t = F_j t_1 \dots t_m$ and, by induction hypothesis, all t_i satisfy $(*)$. Here, we need one additional step:

$$\begin{aligned}
[\![F_j t_1 \dots t_m]\!]_{(\bar{g},\bar{a})} &= g_j([\![t_1]\!]_{(\bar{g},\bar{a})}, \dots, [\![t_m]\!]_{(\bar{g},\bar{a})}) \\
&\leq g_j([\![t_1]\!]_{(\bar{h},\bar{a})}, \dots, [\![t_m]\!]_{(\bar{h},\bar{a})}) \\
&\leq h_j([\![t_1]\!]_{(\bar{h},\bar{a})}, \dots, [\![t_m]\!]_{(\bar{h},\bar{a})}) \\
&= [\![F_j t_1 \dots t_m]\!]_{(\bar{h},\bar{a})}.
\end{aligned}$$

□

Term functionals are not only monotonic. In addition, they preserve limits. We formalize this property, which is crucial for constructing fixed points, by means of continuous functions over complete partial orders.

Definition 22: *Let $\mathfrak{A} = \langle A; \leq \rangle$ be a partial order, $T \subseteq A$, and $a \in A$.*

(i) T is called ***directed*** *if $T \neq \emptyset$ and if for every $a, b \in T$ there exists $c \in T$ such that $\{a, b\} \leq c$.*
(ii) a is called an ***upper bound*** *of T if $T \leq a$.*
(iii) a is called a ***least element*** *of T if $a \leq T$ and $a \in T$.*
(iv) If $\{a \mid T \leq a\}$ has a least element, it is called ***least upper bound*** *of T and is denoted by $\bigsqcup T$ or $\bigsqcup_{t \in T} t$.*
(v) $\mathfrak{A}$ is called a ***complete partial order*** *if the following holds:*
 - *There is a least element $\perp_{\mathfrak{A}} \in A$.*
 - *Every directed subset $T \subseteq A$ has a least upper bound $\bigsqcup T \in A$.*

For a directed subset T it follows that any finite subset $T' \subseteq T$ has an upper bound in T. Hence, if T itself is finite, it must contain an upper bound, necessarily the least one: $\bigsqcup T \in T$. Therefore, a flat partial order $\langle A_\perp; \leq \rangle$ is complete because a directed subset contains at most one $a \in A$. The following lemma states that completeness is preserved under product and function space construction.

Lemma 23: *If $\mathfrak{A}_1 = \langle A_1; \leq_1 \rangle$ and $\mathfrak{A}_2 = \langle A_2; \leq_2 \rangle$ are complete partial orders, the same holds for the product space $\mathfrak{A}_1 \times \mathfrak{A}_2$ and the function space $[\mathfrak{A}_1 \to \mathfrak{A}_2]$. Directed subsets $T \subseteq A_1 \times A_2$ and $D \subseteq \{f \mid f : A_1 \to A_2\}$ have the following least upper bounds:*

(i) $\bigsqcup T = (\bigsqcup \mathsf{proj}_1(T), \bigsqcup \mathsf{proj}_2(T))$ and
(ii) $\bigsqcup D$ where $(\bigsqcup D)(a) := \bigsqcup_{f \in D} f(a)$.

Proof: We know already that product and function spaces inherit ordering relations from their components. Obviously, $(\perp_{\mathfrak{A}_1}, \perp_{\mathfrak{A}_2})$ and $a_1 \mapsto \perp_{\mathfrak{A}_2}$ are their least elements, respectively. If $T \subseteq A_1 \times A_2$ is directed, this must also be true for both $\mathsf{proj}_i(T)$. Hence, $(\bigsqcup \mathsf{proj}_1(T), \bigsqcup \mathsf{proj}_2(T))$ exists and proves to be the least upper bound of T. Similarly, it follows for D that all $\{f(a) \mid f \in D\}$ are directed. So, we get $a \mapsto \bigsqcup_{f \in D} f(a)$ as the least upper bound of D. □

We conclude that the spaces $A_\perp^w$ and $\mathbf{F}^{(w,s)}(A_\perp)$ are complete for any $(w,s) \in F(S)$. The next result shows that this also holds if we restrict ourselves to monotonic functions.

Lemma 24: *The function space* $\mathbf{mF}^{(w,s)}(A_\perp)$ *is a complete partial order for any* $(w,s) \in F(S)$.

Proof: The least element of $\mathbf{F}^{(w,s)}(A_\perp)$, $\bar{a} \mapsto \perp^s$, is a constant function and therefore monotonic, thus also being the least element of $\mathbf{mF}^{(w,s)}(A_\perp)$. If $D \subseteq \mathbf{mF}^{(w,s)}(A_\perp)$ is directed, it is also a directed subset of $\mathbf{F}^{(w,s)}(A_\perp)$ and has a least upper bound $g := \bigsqcup D \in \mathbf{F}^{(w,s)}(A_\perp)$ with $g(\bar{a}) = \bigsqcup_{f \in D} f(\bar{a})$. We show that g is monotonic. Let $\bar{a} \leq \bar{b}$ in $A_\perp^w$. Since each $f \in D$ is monotonic, we have $f(\bar{a}) \leq f(\bar{b})$ so that $\bigsqcup_{f \in D} f(\bar{b})$ is an upper bound of $\{f(\bar{a}) \mid f \in D\}$. Therefore, $g(\bar{a}) = \bigsqcup_{f \in D} f(\bar{a}) \leq \bigsqcup_{f \in D} f(\bar{b}) = g(\bar{b})$. $\square$

Finally, we turn to continuous functions. In particular, we want to verify the continuity of term functionals.

Definition 25: *Let* $\mathfrak{A}_1 = \langle A_1; \leq_1 \rangle$ *and* $\mathfrak{A}_2 = \langle A_2; \leq_2 \rangle$ *be complete partial orders and* $f : A_1 \to A_2$. *Then* f *is called* ***continuous*** *if*

- f *is monotonic and*
- $f(\bigsqcup T) = \bigsqcup f(T)$ *for each directed subset* $T \subseteq A_1$.

As a direct consequence we note that continuous functions are closed under composition.

Lemma 26: *Each* $f \in \mathbf{mF}^{(w,s)}(A_\perp)$ *is continuous.*

Proof: Let $D \subseteq A_\perp^w$ be directed. Then, all $\mathsf{proj}_i(D)$ are directed subsets of $A_\perp^{s_i}$. They must be finite and therefore D, too. As f is monotonic, $f(D)$ is also directed and finite. Hence, $\bigsqcup D = d_1 \in D$ and $\bigsqcup f(D) = f(d_2) \in f(D)$ and thereby $\bigsqcup f(D) = f(d_2) \leq f(\bigsqcup D) = f(d_1) \leq \bigsqcup f(D)$. $\square$

Theorem 27: *Each term functional*

$$[\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp} : \mathbf{mF}^\rho(A_\perp) \to \mathbf{mF}^{(w,s)}(A_\perp)$$

is continuous.

Proof: We have checked already that $[\![\lambda \bar{F}.\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp}$ is monotonic. It remains to prove that for any directed $D \subseteq \mathbf{mF}^\rho(A_\perp)$ and for each $\bar{a} \in A_\perp^w$, it holds that

$$[\![\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp[\bigsqcup D]}(\bar{a}) = \bigsqcup_{\bar{g} \in D} [\![\lambda^\sigma \bar{x}.t]\!]_{\mathfrak{A}_\perp[\bar{g}]}(\bar{a}).$$

In the case that $(a_1, \ldots, a_\sigma) \notin A^\sigma$, we get $\perp^s$ on both sides. Otherwise, the argument $\bar{a}$ is passed to the term semantics, and we have to verify that

$$(*) \qquad [\![t]\!]_{(\mathfrak{A}_\perp[\bigsqcup D],[\bar{x}/\bar{a}])} = \bigsqcup_{\bar{g} \in D} [\![t]\!]_{(\mathfrak{A}_\perp[\bar{g}],[\bar{x}/\bar{a}])}$$

We proceed by induction on t, again using the abbreviation

$$[\![t]\!]_{(\bar{g},\bar{a})} := [\![t]\!]_{(\mathfrak{A}_\perp[\bar{g}],[\bar{x}/\bar{a}])}.$$

(i) $t = x_i$ yields a_i on both sides of $(*)$.

(ii) Let $t = ft_1 \dots t_m$, $f \in F$, and $(*)$ holds for all t_i by induction hypothesis. This implies

$$\begin{aligned}
[\![ft_1 \dots t_m]\!]_{(\bigsqcup D, \bar{a})} &= f_{\mathfrak{A}_\perp}([\![t_1]\!]_{(\bigsqcup D, \bar{a})}, \dots, [\![t_m]\!]_{(\bigsqcup D, \bar{a})}) \\
&= f_{\mathfrak{A}_\perp}(\bigsqcup_{\bar{g} \in D} [\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, \bigsqcup_{\bar{g} \in D} [\![t_m]\!]_{(\bar{g}, \bar{a})}) \\
&= f_{\mathfrak{A}_\perp}(\bigsqcup_{\bar{g} \in D} ([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})})) \\
&= \bigsqcup_{\bar{g} \in D} [\![ft_1 \dots t_m]\!]_{(\bar{g}, \bar{a})}.
\end{aligned}$$

(iii) Let $t = F_j t_1 \dots t_m$, $1 \le j \le r$, and $(*)$ holds for all t_i by induction hypothesis. We conclude

$$\begin{aligned}
[\![F_j t_1 \dots t_m]\!]_{(\bigsqcup D, \bar{a})} &= \mathsf{proj}_j(\bigsqcup D)([\![t_1]\!]_{(\bigsqcup D, \bar{a})}, \dots, [\![t_m]\!]_{(\bigsqcup D, \bar{a})}) \\
&= \mathsf{proj}_j(\bigsqcup D)(\bigsqcup_{\bar{g} \in D} ([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})})) \\
&= \bigsqcup_{\bar{g} \in D} \mathsf{proj}_j(\bigsqcup D)([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \\
&= \bigsqcup_{\bar{g} \in D} (\bigsqcup_{\bar{h} \in D} h_j)([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \\
&= \bigsqcup_{\bar{g} \in D} \bigsqcup_{\bar{h} \in D} h_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \qquad (a) \\
&= \bigsqcup_{\bar{g} \in D} g_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \qquad (b) \\
&= \bigsqcup_{\bar{g} \in D} [\![F_j t_1 \dots t_m]\!]_{(\bar{g}, \bar{a})}.
\end{aligned}$$

All equalities, except the last but one, are obvious. So, let us prove that $(a) = (b)$.

– $(b) \le (a)$ follows from the fact that, for all $\bar{g} \in D$,

$$g_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \le \bigsqcup_{\bar{h} \in D} h_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}).$$

– For the converse $(a) \le (b)$ we observe that for each $\{\bar{g}, \bar{h}\} \subseteq D$ there is $\bar{k} \in D$ such that $\{\bar{g}, \bar{h}\} \le \bar{k}$. We get

$$\begin{aligned}
h_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) &\le k_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \\
&\le k_j([\![t_1]\!]_{(\bar{k}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{k}, \bar{a})})
\end{aligned}$$

so that, for all $\bar{g} \in D$, we have

$$\bigsqcup_{\bar{h} \in D} h_j([\![t_1]\!]_{(\bar{g}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{g}, \bar{a})}) \le \bigsqcup_{\bar{k} \in D} k_j([\![t_1]\!]_{(\bar{k}, \bar{a})}, \dots, [\![t_m]\!]_{(\bar{k}, \bar{a})}),$$

which implies $(a) \le (b)$. □

We conclude our order–theoretic foundations with Tarski's Fixed–Point Theorem for continuous transformations [Tar55].

Theorem 28: *Let $\mathfrak{A} = \langle A; \leq \rangle$ be a complete partial order and $f : A \to A$ a continuous mapping. Then we have for $D := \{f^i(\bot) \mid i \in \mathbb{N}\}$ that*

(i) D *is directed and*
(ii) $\mathsf{fix}(f) := \bigsqcup D$ *is a fixed point of* f*, i.e.,* $f(\mathsf{fix}(f)) = \mathsf{fix}(f)$*, and moreover,*
(iii) $\mathsf{fix}(f)$ *is the least fixed point of* f*.*

Proof: (i) $\bot \leq f(\bot) \leq f^2(\bot) \leq \ldots$ because f is monotonic. Hence, D is directed.
(ii) $f(\bigsqcup_{i\in\mathbb{N}} f^i(\bot)) = \bigsqcup_{i\in\mathbb{N}} f^{i+1}(\bot) = \bigsqcup_{i\in\mathbb{N}} f^i(\bot)$ because f is continuous and $\bot$ is least element.
(iii) For $a \in A$ with $f(a) = a$ it follows that $f^i(\bot) \leq f^i(a) = a$ for every $i \in \mathbb{N}$. Hence, $\mathsf{fix}(f) \leq a$. □

3 Recursive Function Definitions

With these algebraic and order–theoretic foundations we established a suitable framework for defining syntax and semantics of recursive function definitions. In this section we present a fixed–point semantics that takes strictness information into account. As a preparation for our stack implementation we construct an operational small–step semantics. Its soundness and completeness is proved by means of a third semantics, a non–deterministic reduction semantics whose parallel reduction steps turn out to essentially support the equivalence proofs.

Formally, a recursive function definition is viewed as a scheme together with an algebra that interprets its basic function symbols.

Definition 29: *Let $\Sigma = \langle S, F\rangle$ be a branching signature and $\mathbb{F}_\rho = \{F_1^{\tau_1}, \ldots, F_r^{\tau_r}\}$ a non–empty set of function variables with $\rho = \tau_1 \ldots \tau_r \in F(S)^+$. Let $\tau_j = (w_j, s_j)$, $dt_j \in T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X}_{w_j})^{s_j}$, and $0 \leq \sigma_j \leq |w_j|$ for $j = 1, \ldots, r$. Then we call*

$$R = (F_j^{\tau_j} = \lambda^{\sigma_j} x^{w_j}.dt_j \mid 1 \leq j \leq r)$$

a ***recursive function scheme*** *over Σ. If in addition $\mathfrak{A}$ is a branching Σ–algebra, we call $(R, \mathfrak{A})$ a* ***recursive function definition*** *over Σ.*

The sort of a recursive function definition is given by its defining function variable $F_1^{\tau_1}$:

$$\mathsf{sort}(R, \mathfrak{A}) := \tau_1.$$

Thereby, the set of recursive function definitions over Σ, $\mathbf{Rfd}_\Sigma$, is $F(S)$–sorted:

$$\mathbf{Rfd}_\Sigma := \bigcup_{\tau\in F(S)} \mathbf{Rfd}_\Sigma^\tau.$$

Example 30: It should be clear that our introductory example can be understood as a recursive function definition $(R_{mult}, \mathfrak{N}) \in \mathbf{Rfd}_\Sigma$ where Σ is a branching signature with sorts for non–negative integers and booleans and with appropriate function symbols for addition, subtraction, zero, one, equality, and conditionals. Accordingly, $\mathfrak{N}$ fixes their standard interpretation. We represent R_{mult} syntactically by

$$
\begin{aligned}
F &= \lambda^0\, x.G\,(x-1)\,(Hx)\\
G &= \lambda^1\, x\, y.\mathsf{cond}\,(x=0)\,x\,((G\,(x-1)\,y)+y)\\
H &= \lambda^0\, x.H\,(x+1).
\end{aligned}
$$

Here, we have chosen the strictness information such that only the first argument of G is treated strictly which corresponds to the third case of mixed extension in our introduction.

3.1 Denotational Semantics

Definition 31 (Fixed–point semantics): *Let* $(R,\mathfrak{A}) \in \mathbf{Rfd}_{\Sigma}$. *In order to specify its* ***fixed–point semantics*** *as a function of type*

$$\mathsf{Fp}[\![R]\!]_{\mathfrak{A}} : A^{w_1} \to A_{\perp}^{s_1},$$

we associate with $(R,\mathfrak{A})$ *the transformation*

$$\Phi_{(R,\mathfrak{A})} : \mathbf{mF}^{\rho}(A_{\perp}) \to \mathbf{mF}^{\rho}(A_{\perp})$$

given by

$$\Phi_{(R,\mathfrak{A})}(\bar{g}) := ([\![\lambda\bar{F}.\lambda^{\sigma_1}x^{w_1}.dt_1]\!]_{\mathfrak{A}_{\perp}}(\bar{g}),\ldots,[\![\lambda\bar{F}.\lambda^{\sigma_r}x^{w_r}.dt_r]\!]_{\mathfrak{A}_{\perp}}(\bar{g})),$$

and define

$$\mathsf{Fp}[\![R]\!]_{\mathfrak{A}}(a^{w_1}) := b^{s_1} \quad \text{if} \quad \mathsf{proj}_1(\mathsf{fix}(\Phi_{(R,\mathfrak{A})}))(a^{w_1}) = b^{s_1}.$$

Note that solutions of $(R,\mathfrak{A})$ viewed as an equation system are in fact fixed points of $\Phi_{(R,\mathfrak{A})}$. Its least fixed point exists due to the continuity of term functionals (Theorem 27). Since $\perp$–elements are only used as intermediate denotational values, we did not specify the semantics just by $\mathsf{proj}_1(\mathsf{fix}(\Phi_{(R,\mathfrak{A})})) : A_{\perp}^{w_1} \to A_{\perp}^{s_1}$ but by its restriction to A^{w_1} instead.

Example 32: We compute the fixed–point semantics of our example $(R_{mult},\mathfrak{N})$

$$\mathsf{Fp}[\![R_{mult}]\!]_{\mathfrak{N}} : \mathbb{N} \to \mathbb{N}_{\perp}$$

as follows:

The initial approximation is the least element of the domain, i.e., the vector containing a globally undefined function for each of the equations:

$$
\begin{aligned}
f_0(a) &= \perp\\
g_0(a,b) &= \perp\\
h_0(a) &= \perp \qquad \text{for all } a,b \in \mathbb{N}_{\perp}
\end{aligned}
$$

In the first iteration, these initial functions are substituted for function variables on right–hand sides:

$$\begin{aligned}
f_1(a) &= g_0(a-1, h_0(a)) \\
&= \perp \\
g_1(a,b) &= \begin{cases} \perp & \text{if } a = \perp \\ 0 & \text{if } a = 0 \\ g_0(a-1,b)+b & \text{otherwise} \end{cases} \\
&= \begin{cases} 0 & a = 0 \\ \perp & \text{otherwise} \end{cases} \\
h_1(a) &= h_0(a+1) \\
&= \perp
\end{aligned}$$

The next iteration yields:

$$\begin{aligned}
f_2(a) &= \begin{cases} 0 & \text{if } a \in \{0,1\} \\ \perp & \text{otherwise} \end{cases} \\
g_2(a,b) &= \begin{cases} 0 & \text{if } a = 0 \\ b & \text{if } a = 1 \\ \perp & \text{otherwise} \end{cases} \\
h_2(a) &= \perp
\end{aligned}$$

Continuing this process and taking the least upper bound of the resulting approximations we get as least fixed point:

$$\begin{aligned}
f(a) &= \begin{cases} 0 & \text{if } a \in \{0,1\} \\ \perp & \text{otherwise} \end{cases} \\
g(a,b) &= \begin{cases} 0 & \text{if } a = 0 \\ a \cdot b & \text{if } a, b \neq \perp \\ \perp & \text{otherwise} \end{cases} \\
h(a) &= \perp
\end{aligned}$$

and therefore:

$$\mathsf{Fp}[\![R_{mult}]\!]_{\mathfrak{N}}(a) = \begin{cases} 0 & \text{if } a \in \{0,1\} \\ \perp & \text{if } a > 1 \end{cases}$$

3.2 Operational Semantics

As a first step towards an implementation on an abstract stack machine, we change our view of recursive function definitions by taking an operational perspective in which equations are regarded as rewrite rules. They allow to compute a function value from given arguments by stepwise reduction.

Let $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$. Based on the S–sorted set

$$\mathbf{T}(R, \mathfrak{A}) := T_{\Sigma[\mathbb{F}_\rho]}(A)$$

of **reduction terms** for $(R, \mathfrak{A})$, we first define the following **computation rules**:

- **ground reduction:** $fa^w \to f_{\mathfrak{A}}(a^w)$
  for every $(w,s) \in F(S)$, $f \in F^{(w,s)}_{\mathsf{base}}$, and $a^w \in A^w$,
- **conditional reduction:** $\mathsf{cond}_s\,\mathsf{true}\,u_1u_2 \to u_1$
  $\mathsf{cond}_s\,\mathsf{false}\,u_1u_2 \to u_2$
  for every $s \in S$ and $u_1, u_2 \in \mathbf{T}(R,\mathfrak{A})^s$, and
- **function reduction:** $F_j^{\tau_j} u^{w_j} \to dt_j[x^{w_j}/u^{w_j}]$
  for every $j = 1, \ldots, r$ and $u^{w_j} \in \mathbf{T}(R,\mathfrak{A})^{w_j}$ such that $u_1, \ldots, u_{\sigma_j} \in A$.

Note that in the special case $w = \varepsilon$ ground reduction yields constant reduction: $c \to c_{\mathfrak{A}}()$. Also note that function reduction respects the strictness information since the first σ_j arguments have to be evaluated before the function call.

Generally, a reduction term contains several reducible subterms so that a sequential implementation requires a suitable reduction strategy. Later we shall see that leftmost reduction represents an appropriate choice. At this point, however, we prefer to leave this issue open, even permitting parallel reduction steps, in order to simplify the correctness proof.

Definition 33: *The* ***reduction relation***

$$\Rightarrow\ \subseteq\ \mathbf{T}(R,\mathfrak{A}) \times \mathbf{T}(R,\mathfrak{A})$$

is inductively defined as follows:

- *For each computation rule $u \to v$, we have $u \Rightarrow v$.*
- *For each $u \in \mathbf{T}(R,\mathfrak{A})$, we let $u \Rightarrow u$.*
- *For each $(w,s) \in F(S)$, $\varphi \in (F \cup \mathbb{F}_\rho)^{(w,s)}$, and $u^w, v^w \in \mathbf{T}(R,\mathfrak{A})^w$ such that $u_i \Rightarrow v_i$ for every $1 \le i \le |w|$, we have $\varphi u^w \Rightarrow \varphi v^w$.*

Example reductions can be found in the introduction.

The following lemma states that we can in fact reduce arbitrary subterms simultaneously.

Lemma 34: *Let $w \in S^+$, $t \in T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X}_w)$, and $u^w, v^w \in \mathbf{T}(R,\mathfrak{A})^w$. If $u_i \Rightarrow v_i$ for every $1 \le i \le |w|$, then also*

$$t[x^w/u^w] \Rightarrow t[x^w/v^w].$$

Proof: by induction on $t \in T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X}_w)$.

(i) If $t = x_i$, then the assertion turns into one of the premises.
(ii) Let $t = \varphi t_1 \ldots t_m$, and let as induction hypothesis the assertion hold for every t_i. It follows that

$$\begin{aligned}(\varphi t_1 \ldots t_m)[x^w/u^w] &= \varphi(t_1[x^w/u^w]) \ldots (t_m[x^w/u^w])\\ &\Rightarrow \varphi(t_1[x^w/v^w]) \ldots (t_m[x^w/v^w])\\ &= (\varphi t_1 \ldots t_m)[x^w/v^w].\end{aligned}$$ □

Despite its nondeterminism, the reduction relation provides a proper semantics as it proves to be confluent: for a given reduction term $u \in \mathbf{T}(R,\mathfrak{A})$, there exists at most one value $a \in A$ such that $u \Rightarrow^* a$. This will be verified by extending the fixed–point semantics to reduction terms, and by establishing its invariance under reduction. For notational convenience, we introduce some abbreviations concerning the fixed point of the transformation $\Phi_{(R,\mathfrak{A})}$. Let

$$\perp^\rho := (\perp_1^{\tau_1}, \ldots, \perp_r^{\tau_r}) := \perp_{\mathbf{mF}^\rho(A_\perp)}$$

where $\perp^{(w,s)} : A_\perp^w \to A_\perp^s$ is given by

$$\perp^{(w,s)}(a^w) := \perp^s$$

for every $a^w \in A_\perp^w$. We denote the fixed point of $\Phi_{(R,\mathfrak{A})}$ by

$$\psi^\rho = (\psi_1^{\tau_1}, \ldots, \psi_r^{\tau_r}) := \mathsf{fix}(\Phi_{(R,\mathfrak{A})})$$

or, omitting type information, simply by $\bar{\psi} = (\psi_1, \ldots, \psi_r)$, and its approximations by

$$\bar{\psi}^{(k)} = (\psi_1^{(k)}, \ldots, \psi_r^{(k)}) := \Phi_{(R,\mathfrak{A})}^k(\perp^\rho)$$

for each $k \in \mathbb{N}$.

Now we extend our fixed–point semantics to a reduction term $u \in \mathbf{T}(R, \mathfrak{A})$ by

$$\mathsf{Fp}[\![u]\!]_{(R,\mathfrak{A})} := [\![u]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}$$

which relates to the fixed–point semantics of a recursive function definition as follows:

$$\mathsf{Fp}[\![R]\!]_{\mathfrak{A}}(a^w) = \mathsf{Fp}[\![F_1 a^w]\!]_{(R,\mathfrak{A})}$$

for every $a^w \in A^w$.

Lemma 35: *For every $u, v \in \mathbf{T}(R, \mathfrak{A})$, $u \Rightarrow v$ implies* $\mathsf{Fp}[\![u]\!]_{(R,\mathfrak{A})} = \mathsf{Fp}[\![v]\!]_{(R,\mathfrak{A})}$.

Proof: by induction on the structure of $\Rightarrow$.

(i) For a ground reduction of the form $f a^w \to f_{\mathfrak{A}}(a^w)$, it follows directly that

$$[\![f a^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]} = f_{\mathfrak{A}}(a^w) = [\![f_{\mathfrak{A}}(a^w)]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}.$$

(ii) For a conditional reduction $\mathsf{cond}_s\,\mathsf{true}\,u_1 u_2 \to u_1$ we have

$$\begin{aligned}[\![\mathsf{cond}_s\,\mathsf{true}\,u_1 u_2 \to u_1]\!]_{\mathfrak{A}_\perp[\bar{\psi}]} &= \alpha_\perp(\mathsf{cond}_s)(\mathsf{true}, [\![u_1]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}), [\![u_2]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}) \\ &= [\![u_1]\!]_{\mathfrak{A}_\perp[\bar{\psi}]},\end{aligned}$$

and correspondingly in the false–case.

(iii) A function reduction $F_j u^w \to dt_j[x^w/u^w]$ with $u^w = (u_1, \ldots, u_n)$ requires that $u_1, \ldots, u_{\sigma_j} \in A$. Here the assertion follows essentially from the fixed–point property of $\bar{\psi}$ and from Lemma 11:

$$\begin{aligned}[\![F_j u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]} &= \psi_j([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}) \\ &= \mathsf{proj}_j(\bar{\psi})([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}) \\ &= \mathsf{proj}_j(\Phi_{(R,\mathfrak{A})}(\bar{\psi}))([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}) \\ &= [\![\lambda F^\rho.\lambda^{\sigma_j} x^w.dt_j]\!]_{\mathfrak{A}_\perp}(\bar{\psi})([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}) \\ &= [\![\lambda^{\sigma_j} x^w.dt_j]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}) \\ &= [\![dt_j]\!]_{(\mathfrak{A}_\perp[\bar{\psi}],[x^w/[\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}])} \\ &= [\![dt_j[x^w/u^w]]\!]_{\mathfrak{A}_\perp[\bar{\psi}]}.\end{aligned}$$

(iv) For the inductive step, let $\varphi u_1 \dots u_n \Rightarrow \varphi v_1 \dots v_n$ with $u_i \Rightarrow v_i$ for every $1 \le i \le n$. We easily check that

$$\begin{aligned}
[\![\varphi u_1 \dots u_n]\!]_{\mathfrak{A}_\perp[\bar\psi]} &= \varphi_{\mathfrak{A}_\perp[\bar\psi]}([\![u_1]\!]_{\mathfrak{A}_\perp[\bar\psi]}, \dots, [\![u_n]\!]_{\mathfrak{A}_\perp[\bar\psi]}) \\
&= \varphi_{\mathfrak{A}_\perp[\bar\psi]}([\![v_1]\!]_{\mathfrak{A}_\perp[\bar\psi]}, \dots, [\![v_n]\!]_{\mathfrak{A}_\perp[\bar\psi]}) \\
&= [\![\varphi v_1 \dots v_n]\!]_{\mathfrak{A}_\perp[\bar\psi]},
\end{aligned}$$

thus concluding the proof. □

The invariance of the fixed–point semantics entails the following special confluence property, which yields the well–definedness of reduction semantics as specified below.

Corollary 36: *For every $u \in \mathbf{T}(R, \mathfrak{A})$ and $b_1, b_2 \in A$, $u \Rightarrow^* b_1$ and $u \Rightarrow^* b_2$ implies $b_1 = b_2$.*

Definition 37 (Reduction semantics): *Let $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma^{(w,s)}$. Its* ***reduction semantics***

$$\mathsf{Red}[\![R]\!]_{\mathfrak{A}} : A^w \to A^s_\perp,$$

is defined for $a^w \in A^w$ by:

$$\mathsf{Red}[\![R]\!]_{\mathfrak{A}}(a^w) := \begin{cases} b & \text{if } F_1 a^w \Rightarrow^* b \text{ for some } b \in A^s \\ \perp^s & \text{if no such } b \text{ exists} \end{cases}$$

Moreover, the invariance of fixed–point semantics yields the following soundness property.

Corollary 38 (Soundness of reduction semantics): *For $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$, $a^w \in A^w$, and $b \in A^s$, it holds that*

$$\mathsf{Red}[\![R]\!]_{\mathfrak{A}}(a^w) = b \quad \textit{implies} \quad \mathsf{Fp}[\![R]\!]_{\mathfrak{A}}(a^w) = b.$$

It remains to verify that the reduction semantics is also complete with respect to the fixed–point semantics. Here we exploit the fact that the restriction to finite approximations is sufficient to obtain the denotational semantics of a given term.

Lemma 39: *For each $u \in \mathbf{T}(R, \mathfrak{A})$ there exists $k \in \mathbb{N}$ such that*

$$[\![u]\!]_{\mathfrak{A}_\perp[\bar\psi]} = [\![u]\!]_{\mathfrak{A}_\perp[\bar\psi^{(k)}]}.$$

In particular, if $[\![u]\!]_{\mathfrak{A}_\perp[\bar\psi]} = \perp^s$, this holds with $k = 0$.

Proof: From the proof of continuity for term functionals (Theorem 27) we see that

$$[\![u]\!]_{\mathfrak{A}_\perp[\bar\psi]} = \bigsqcup_{k \in \mathbb{N}} [\![u]\!]_{\mathfrak{A}_\perp[\bar\psi^{(k)}]}$$

because $\{\bar\psi^{(k)} \mid k \in \mathbb{N}\}$ is a directed subset of $\mathbf{mF}^\rho(A_\perp)$ with $\bigsqcup_{k\in\mathbb{N}} \bar\psi^{(k)} = \bar\psi$, and u can be regarded as a term in $T_{\Sigma'[\mathbb{F}_\rho]}$ where Σ' is a suitable extension of Σ by constant symbols from A. As we have $\bigsqcup_{k\in\mathbb{N}} [\![u]\!]_{\mathfrak{A}_\perp[\bar\psi^{(k)}]} \in A_\perp$, there must be a $k \in \mathbb{N}$ satisfying the assertion. Obviously, if $\bigsqcup_{k\in\mathbb{N}} [\![u]\!]_{\mathfrak{A}_\perp[\bar\psi^{(k)}]} = \perp^s$, we can take $k = 0$. □

Theorem 40 (Completeness of reduction semantics): *Let $u \in \mathbf{T}(R,\mathfrak{A})$, $a \in A$, and $k \in \mathbb{N}$. Then*

$$(*) \qquad [\![u]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k)}]} = a \quad \textit{implies} \quad u \Rightarrow^* a.$$

Proof: by induction on $k \in \mathbb{N}$.

(i) $k = 0$: We prove $(*)$ by induction on the structure of $u \in \mathbf{T}(R,\mathfrak{A})$.

(a) $u = a \in A$: $(*)$ obviously holds since $a \Rightarrow^* a$.

(b) $u = fu_1 \dots u_n$ with $f \in F_{\mathsf{base}}$: let $(*)$ hold for every u_i, where $k = 0$. Then it follows that

$$\begin{aligned} a &= [\![fu_1 \dots u_n]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} \\ &= f_{\mathfrak{A}_\perp}([\![u_1]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]}, \dots, [\![u_n]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]}). \end{aligned}$$

Due to the strictness of $f_{\mathfrak{A}_\perp}$, for every argument position $i \in \{1, \dots, n\}$ there exists $a_i \in A$ such that $[\![u_i]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = a_i$, and hence, according to the induction hypothesis, $u_i \Rightarrow^* a_i$, which implies $fu_1 \dots u_n \Rightarrow^* fa_1 \dots a_n \Rightarrow a$.

(c) $u = \mathsf{cond}\, u_0 u_1 u_2$ with $(*)$ for every u_i, where $k = 0$. Here, $[\![u]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = a$ implies that either $[\![u_0]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = \mathsf{true}$ and $[\![u_1]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = a$ or $[\![u_0]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = \mathsf{false}$ and $[\![u_2]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = a$. In both cases, the induction hypothesis yields reduction sequences which can be combined to $\mathsf{cond}\, u_0 u_1 u_2 \Rightarrow^* a$.

(d) $u = F_j u^w$: here, $[\![u]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]} = \psi_j^{(0)}([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(0)}]}) = \perp^{s_j} \notin A$, such that $(*)$ holds trivially.

Altogether, $(*)$ holds for $k = 0$.

(ii) $k \rightsquigarrow k+1$: let $(*)$ hold for k. Again we employ induction on the structure of $u \in \mathbf{T}(R,\mathfrak{A})$. The first three cases, which do not directly depend on the approximation index k, can be handled in analogy to $k = 0$. It remains to investigate the following situation:

(d) $u = F_j u^w$ where $u^w = (u_1, \dots, u_n)$. Our structural induction hypothesis yields $(*)$ with $k+1$ for every u_i, which implies

$$\begin{aligned} a &= [\![u]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]} \\ &= \psi_j^{(k+1)}([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]}) \\ &= \mathsf{proj}_j(\Phi_{(R,\mathfrak{A})}(\bar{\psi}^{(k)}))([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]}) \\ &= [\![\lambda^{\sigma_j} x^w . dt_j]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k)}]}([\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]}) \\ &= [\![dt_j]\!]_{(\mathfrak{A}_\perp[\bar{\psi}^{(k)}],[x^w/[\![u^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]}])}. \end{aligned}$$

Here, the strictness index σ_j requires that $a_i := [\![u_i]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]} \in A$ for every $i = 1, \dots, \sigma_j$. In order to be able to apply the induction hypothesis for k, we choose the reduction terms

$$v_i := \begin{cases} a_i & \text{if } a_i \in A \\ u_i & \text{if } a_i = \perp^{s_i} \end{cases}$$

for every $i = 1, \dots, n$. It follows that $[\![v_i]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k)}]} = [\![u_i]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]}$, since the monotonicity of the term semantics implies that $[\![u_i]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k)}]} \leq [\![u_i]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k+1)}]}$. Applying the Substitution Lemma, we obtain

$$\begin{aligned} a &= [\![dt_j]\!]_{(\mathfrak{A}_\perp[\bar{\psi}^{(k)}],[x^w/[\![v^w]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k)}]}])} \\ &= [\![dt_j[x^w/v^w]]\!]_{\mathfrak{A}_\perp[\bar{\psi}^{(k)}]} \end{aligned}$$

such that the induction hypothesis for k yields

$$dt_j[x^w/v^w] \Rightarrow^* a.$$

Moreover, v^w has been chosen to fulfil the strictness requirement of F_j, and hence

$$F_j v^w \Rightarrow dt_j[x^w/v^w].$$

Finally, it is easily verified that $u_i \Rightarrow^* v_i$ for every $i = 1, \dots, n$ such that

$$F_j u^w \Rightarrow^* a,$$

which concludes our proof. □

Combining Corollary 38 and Theorem 40, we obtain the following equivalence result.

Corollary 41 (Equivalence of reduction and fixed–point semantics): *For every $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ we have*

$$\mathsf{Red}[\![R]\!]_{\mathfrak{A}} = \mathsf{Fp}[\![R]\!]_{\mathfrak{A}}.$$

3.3 Leftmost Reduction

Due to its nondeterminism, the reduction semantics as presented in the previous section is not a suitable basis for a direct implementation. We therefore introduce a deterministic evaluation strategy. As mentioned earlier, selecting the leftmost reducible subterm of a reduction term represents an appropriate choice. We thereby generalize and unify the well–known leftmost–outermost and leftmost–innermost reduction strategies.

Definition 42: *The **l–reduction relation***

$$\Rightarrow_l \ \subseteq \ \mathbf{T}(R, \mathfrak{A}) \times \mathbf{T}(R, \mathfrak{A})$$

is inductively defined as follows.

- *For each computation rule $u \rightarrow v$, we have $u \Rightarrow_l v$.*
- *If $f a_1 \dots a_{i-1} u_i \dots u_n \in \mathbf{T}(R, \mathfrak{A})$ where $f \in F_{\mathsf{base}}$, $1 \leq i \leq n$, $a_1, \dots, a_{i-1} \in A$, and $u_i \Rightarrow_l v_i$ for some $v_i \in \mathbf{T}(R, \mathfrak{A})$, then*

$$f a_1 \dots a_{i-1} u_i \dots u_n \Rightarrow_l f a_1 \dots a_{i-1} v_i \dots u_n.$$

- *If $\mathsf{cond}_s\, u_0 u_1 u_2 \in \mathbf{T}(R, \mathfrak{A})$ and $u_0 \Rightarrow_l v_0$ for some $v_0 \in \mathbf{T}(R, \mathfrak{A})$, then*

$$\mathsf{cond}_s\, u_0 u_1 u_2 \Rightarrow_l \mathsf{cond}_s v_0 u_1 u_2.$$

- *If $F_j a_1 \dots a_{i-1} u_i \dots u_n \in \mathbf{T}(R, \mathfrak{A})$ where $1 \leq i \leq \sigma_j$, $a_1, \dots, a_{i-1} \in A$, and $u_i \Rightarrow_l v_i$ for some $v_i \in \mathbf{T}(R, \mathfrak{A})$, then*

$$F_j a_1 \dots a_{i-1} u_i \dots u_n \Rightarrow_l F_j a_1 \dots a_{i-1} v_i \dots u_n.$$

Lemma 43: *For every $u \in \mathbf{T}(R, \mathfrak{A})$, we have:*

(i) If $u \notin A$, then there exists exactly one $v \in \mathbf{T}(R, \mathfrak{A})$ such that $u \Rightarrow_l v$.

(ii) If $u \in A$, then no such v exists.

Proof: by induction on the structure of u. □

Definition 44 (l–reduction semantics): *Let $(R, \mathfrak{A}) \in \mathbf{Rfd}_{\Sigma}^{(w,s)}$. We define its* ***l–reduction semantics***

$$\mathsf{lRd}[\![R]\!]_{\mathfrak{A}} : A^w \to A^s_{\perp},$$

by:

$$\mathsf{lRd}[\![R]\!]_{\mathfrak{A}}(a^w) := \begin{cases} b & \text{if } F_1 a^w \Rightarrow_l^* b \text{ for some } b \in A^s \\ \perp^s & \text{if no such } b \text{ exists} \end{cases}$$

Theorem 45 (Equivalence of l–reduction and reduction semantics): *For every $(R, \mathfrak{A}) \in \mathbf{Rfd}_{\Sigma}$ we have*

$$\mathsf{lRd}[\![R]\!]_{\mathfrak{A}} = \mathsf{Red}[\![R]\!]_{\mathfrak{A}}.$$

Proof: Obviously, $F_1 a^w \Rightarrow_l^* b$ implies $F_1 a^w \Rightarrow^* b$. To verify the completeness of l–reduction semantics, we prove by induction on $m \in \mathbb{N}$ that, for every $u \in \mathbf{T}(R, \mathfrak{A})$ and $a \in A$, $u \Rightarrow_l^* a$ whenever $u \Rightarrow^m a$.

(i) $m = 0$: $u \Rightarrow^0 a$ implies $u = a$ and hence $u \Rightarrow_l^* a$.

(ii) $m \rightsquigarrow m+1$: let $u \Rightarrow^{m+1} a$. We have the following cases for u:

(a) $u = a$: as before, $u \Rightarrow_l^* a$ follows immediately.

(b) $u = f u_1 \dots u_n$ with $f \in F_{\mathsf{base}}$:
It follows that $u \Rightarrow^m f a_1 \dots a_n \Rightarrow f_{\mathfrak{A}}(a_1, \dots, a_n) = a$. According to Definition 33, there are n simultaneous reductions of the form $u_i \Rightarrow^m a_i$ such that by induction hypothesis $u_i \Rightarrow_l^* a_i$ for $i = 1, \dots, n$. Sequentially composing these reductions, we obtain

$$f u_1 \dots u_n \Rightarrow_l^* f a_1 \dots u_n \Rightarrow_l^* f a_1 \dots a_n \Rightarrow_l a.$$

(c) $u = \mathsf{cond}\, u_0 u_1 u_2$: analogously

(d) $u = F_j u^w$ with $u^w = (u_1, \dots, u_n)$:
The reduction $u \Rightarrow^{m+1} a$ can be decomposed into $F_j u^w \Rightarrow^p F_j v^w \Rightarrow dt_j[x^w/v^w] \Rightarrow^q a$ where $p + q = m$, $v^w = (v_1, \dots, v_n) \in \mathbf{T}(R, \mathfrak{A})^n$, and $v_1, \dots, v_{\sigma_j} \in A$. Consequently, $u_i \Rightarrow^p v_i$ for $i = 1, \dots, n$, and by induction hypothesis $u_i \Rightarrow_l^* v_i$ for $i = 1, \dots, \sigma_j$. By proper combination we get

$$\begin{aligned} F_j u^w &\Rightarrow_l^* F_j v_1 \dots v_{\sigma_j} u_{\sigma_j+1} \dots u_n \\ &\Rightarrow_l dt_j[x^w/(v_1, \dots, v_{\sigma_j}, u_{\sigma_j+1}, \dots, u_n)] \\ &=: t'. \end{aligned}$$

Now, Lemma 34 admits simultaneous reduction of subterms so that we get $t' \Rightarrow^p dt_j[x^w/v^w] \Rightarrow^q a$, and again by induction hypothesis, $t' \Rightarrow_l^* a$. □

These equivalence results show that strictness information can appropriately be modelled on the denotational as well as the operational level.

4 Interpreter Semantics

In this section we present an interpreter for evaluating a recursive function definition. In comparison with reduction semantics, its working principle is closer to a real implementation: in function calls, variable substitutions are not carried out but, for reasons of efficiency, argument values are kept in environments.

More concretely, our machine consists of three stack components. The first one is used to drive the reduction process according to the structure of the current term. Either, special decomposition steps are carried out to implement the leftmost reduction strategy, or certain computation steps corresponding to computation rules of reduction semantics (cf. Section 3.2) are taken. Here, special constructor symbols are used to delay the evaluation of non–strict function parameters and of the two result branches of a conditional expression.

The second component is a data stack which is used for storing intermediate computation results. The third stack keeps track of parameter values for function calls. It contains one entry for each active function call and for each evaluation of a lazy argument. Special ret entries are used to delete the topmost environment when terminating the corresponding computation.

Note that the standard implementation technique for lazy functional languages involves so–called *closures* which are special data structures used to represent unevaluated arguments (or, in a higher–order setting, partial function applications), and which are usually stored in a heap. This technique was introduced by P. Landin for his SECD–machine [Lan64]. However, our first–order framework without (explicit) data structures allows us to keep all required information in a single nested stack.

As before, let $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ where $R = (F_j^{\tau_j} = \lambda^{\sigma_j} x^{w_j}.dt_j \mid 1 \leq j \leq r)$. First we construct from R two symbol sets for proper control of interpreter steps. An arbitrary term $t \in T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X})$ is used as a decomposition symbol which will be transformed into a sequence of reduction symbols. To ease the following compiler construction we even include atomic terms as decomposition symbols and distinguish them from corresponding reduction symbols although this separation is not necessary.

Definition 46 (Decomposition and reduction symbols): *The elements of* $\mathbf{Dec} := T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X})$ *are called* ***decomposition symbols*** *of* R*. With each* $t \in \mathbf{Dec}$ *we associate a reduction symbol* $\mathsf{redsym}(t)$*:*

- $\mathsf{redsym}(x) := [x]$
- $\mathsf{redsym}(f t_1 \dots t_n) := [f]$
- $\mathsf{redsym}(\mathsf{cond}\, t_0 t_1 t_2) := \mathsf{cond}[t_1, t_2]$
- $\mathsf{redsym}(F_j t_1 \dots t_{n_j}) := F_j[t_{\sigma_j+1}, \dots, t_{n_j}]$

and, including the special reduction symbol ret*, we get the set* $\mathbf{Red} := \{\mathsf{redsym}(t) \mid t \in \mathbf{Dec}\} \cup \{\mathsf{ret}\}$ *of* ***reduction symbols*** *of* R*.*

Now we can fix the interpreter states as follows.

Definition 47: *The set* $\mathbf{IntSt}$ *of* ***interpreter states*** *w.r.t.* $(R, \mathfrak{A})$ *is given by*

$$\mathbf{IntSt} := \mathbf{PS} \times \mathbf{DS} \times \mathbf{ES}$$

where

$$\mathbf{PS} := (\mathbf{Dec} \cup \mathbf{Red})^*,$$
$$\mathbf{DS} := A^*, \textit{ and}$$
$$\mathbf{ES} := \mathbf{Env}^*$$

represent the sets of ***program stack values****,* ***data stack values****, and* ***environment stack values****, respectively. Here,* $\mathbf{Env} := (Z^*)^*$ *denotes the set of* ***environments*** *where* $Z := A \cup \mathbf{Dec}$.

We use the following standard denotations:

$$\begin{array}{ll} st = \langle ps, ds, es\rangle & \in \mathbf{IntSt} \\ es = e_1 : \ldots : e_n & \in \mathbf{ES} \\ e_i = \bar{z_1} \cdot \ldots \cdot \bar{z_m} & \in \mathbf{Env} \\ \bar{z_i} = (z_1, \ldots, z_k) & \in Z^* \end{array}$$

Later we shall restrict states to l–states and, in particular, environments to l–environments, taking type information into account. As a consequence this will also enable us to simplify our notation for environments, which currently represents the full stack history at each stack level.

The interpreter changes its states by performing transitions which are determined by the top symbol of the program stack. The essential point here is the handling of a function call, which extends the environment stack by a new environment, whereas for evaluating a lazy argument the environment of the calling function has to be restored.

Definition 48 (Interpreter transitions): *A state* $st = \langle \varepsilon, ds, es\rangle \in \mathbf{IntSt}$ *is called a* ***final state****. Non–final states are called* ***decomposition states*** *or* ***reduction states*** *according to the top symbol in the program stack. Correspondingly, the* ***transition relation*** $\vdash \subseteq \mathbf{IntSt} \times \mathbf{IntSt}$ *is given by*

$$\vdash := \vdash_{\mathsf{dec}} \cup \vdash_{\mathsf{red}}$$

where the ***decomposition relation*** $\vdash_{\mathsf{dec}}$ *is specified by*

- $\langle x : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle [x] : ps, ds, es\rangle$
- $\langle f t_1 : \ldots : t_n : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle t_1 : \ldots : t_n : [f] : ps, ds, es\rangle$ *for every* $f \in F_{\mathsf{base}}$*,*
- $\langle \mathsf{cond}\, t_0\, t_1\, t_2 : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle t_0 : \mathsf{cond}[t_1, t_2] : ps, ds, es\rangle$*, and*
- $\langle F_j\, t_1 \ldots t_{n_j} : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle t_1 : \ldots : t_{\sigma_j} : F_j[t_{\sigma_j+1}, \ldots, t_{n_j}] : ps, ds, es\rangle$*,*

and where the ***reduction relation*** $\vdash_{\mathsf{red}}$ *is defined by*

- $\langle [x_i] : ps, ds, (z_1, \ldots, z_k) \cdot e : es\rangle \vdash_{\mathsf{red}} \langle ps, ds : z_i, (z_1, \ldots, z_k) \cdot e : es\rangle$
  if $1 \le i \le k$ *and* $z_i \in A$*,*
- $\langle [x_i] : ps, ds, (z_1, \ldots, z_k) \cdot e : es\rangle \vdash_{\mathsf{red}} \langle z_i : \mathsf{ret} : ps, ds, e : (z_1, \ldots, z_k) \cdot e : es\rangle$
  if $1 \le i \le k$ *and* $z_i \in \mathbf{Dec}$*,*
- $\langle [f] : ps, ds : a_1 : \ldots : a_n, es\rangle \vdash_{\mathsf{red}} \langle ps, ds : f_{\mathfrak{A}}(a_1, \ldots, a_n), es\rangle$*,*
- $\langle \mathsf{cond}[t_1, t_2] : ps, ds : \mathsf{true}, es\rangle \vdash_{\mathsf{red}} \langle t_1 : ps, ds, es\rangle$*,*
- $\langle \mathsf{cond}[t_1, t_2] : ps, ds : \mathsf{false}, es\rangle \vdash_{\mathsf{red}} \langle t_2 : ps, ds, es\rangle$*,*
- $\langle F_j[t_{\sigma_j+1}, \ldots, t_{n_j}] : ps, ds : a_1 : \ldots : a_{\sigma_j}, e : es\rangle \vdash_{\mathsf{red}}$
  $\langle dt_j : \mathsf{ret} : ps, ds, (a_1, \ldots, a_{\sigma_j}, t_{\sigma_j+1}, \ldots, t_{n_j}) \cdot e : e : es\rangle$*, and*

– $\langle \mathsf{ret}\!:\!ps, ds, e\!:\!es\rangle \vdash_{\mathsf{red}} \langle ps, ds, es\rangle$.

Note that the transition relation is in fact well–defined because for $st \in \mathbf{IntSt}$ and $st \vdash st'$ we also have $st' \in \mathbf{IntSt}$. Moreover, $\vdash$ is deterministic because each program stack value determines at most one possible transition except those with $[x_i]$ or $\mathsf{cond}[t_1, t_2]$ as top entry, in which case the environment stack or the data stack, respectively, takes the decision.

By successive decomposition a decomposition symbol can be transformed into a sequence of reduction symbols.

Definition 49 (Decomposition mapping): *The* ***decomposition mappping***

$$\mathsf{dec} : \mathbf{PS} \to \mathbf{PS}$$

is given by:

$$\begin{aligned}
\mathsf{dec}(\varepsilon) &:= \varepsilon \\
\mathsf{dec}(p\!:\!ps) &:= \mathsf{dec}(p)\!:\!ps \\
\mathsf{dec}(x) &:= [x] \\
\mathsf{dec}(f t_1 \dots t_n) &:= \mathsf{dec}(t_1)\!:\!t_2\!:\!\dots\!:\!t_n\!:\![f] \\
\mathsf{dec}(\mathsf{cond}\, t_0 t_1 t_2) &:= \mathsf{dec}(t_0)\!:\!\mathsf{cond}[t_1, t_2] \\
\mathsf{dec}(F_j t_1 \dots t_{n_j}) &:= \mathsf{dec}(t_1)\!:\!t_2\!:\!\dots\!:\!t_{\sigma_j} : F_j[t_{\sigma_j+1}, \dots, t_{n_j}] \\
\mathsf{dec}(p) &:= p \quad \textit{if} \quad p \in \mathbf{Red}
\end{aligned}$$

Lemma 50 (Uniqueness of decomposition): *For each non–final state* $st = \langle p\!:\!ps, ds, es\rangle \in \mathbf{IntSt}$ *there is exactly one reduction state* st' *such that* $st \vdash^*_{\mathsf{dec}} st'$. *Moreover,* st' *is the decomposed state* $\mathsf{dec}(st) := \langle \mathsf{dec}(p\!:\!ps), ds, es\rangle$.

Proof: For each $st = \langle p\!:\!ps, ds, es\rangle \in \mathbf{IntSt}$ with $p \in \mathbf{Red}$ we have $\mathsf{dec}(st) = st$, which is a reduction state being reachable within 0 steps. Otherwise we proceed by induction over $p \in \mathbf{Dec}$:

(i) For $p = x$ we have $\langle x\!:\!ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle [x]\!:\!ps, ds, es\rangle = \langle \mathsf{dec}(x)\!:\!ps, ds, es\rangle$.

(ii) If $p = f t_1 \dots t_n$, the induction hypothesis yields:

$$\begin{aligned}
\langle f t_1 \dots t_n\!:\!ps, ds, es\rangle &\vdash_{\mathsf{dec}} \langle t_1\!:\!t_2\!:\!\dots\!:\!t_n\!:\![f]\!:\!ps, ds, es\rangle \\
&\vdash^*_{\mathsf{dec}} \langle \mathsf{dec}(t_1)\!:\!t_2\!:\!\dots\!:\!t_n\!:\![f]\!:\!ps, ds, es\rangle \\
&= \langle \mathsf{dec}(f t_1 \dots t_n)\!:\!ps, ds, es\rangle
\end{aligned}$$

The remaining cases are handled similarly. Note that st' is unique since $\vdash$ is deterministic. □

Definition 51 (Interpreter semantics): *For* $(R, \mathfrak{A}) \in \mathbf{Rfd}^{(w,s)}_{\Sigma}$ *we define its* ***interpreter semantics***

$$\mathsf{Int}[\![R]\!]_{\mathfrak{A}} : A^w \to A^s_{\perp}$$

by:

$$\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(a_1, \dots, a_n) := \begin{cases} b & \textit{if } \langle F_1\, x_1 \dots x_n\!:\!\mathsf{ret}, \varepsilon, (a_1, \dots, a_n)\rangle \vdash^* \langle \varepsilon, b, \varepsilon\rangle \\ \perp^s & \textit{if no such } b \in A^s \textit{ exists} \end{cases}$$

Note that due to the determinism of the transition relation the semantics is well defined.

Example 52: The following computation of our interpreter for the multiplication definition $(R_{mult}, \mathfrak{N})$ (Example 30) shows that

$$\mathsf{Int}[\![R_{mult}]\!]_{\mathfrak{N}}(1) = 0:$$

$$
\begin{array}{l}
\langle F\,x : \mathsf{ret}, \varepsilon, (1)\rangle \\
\vdash_{\mathsf{dec}} \langle F[x] : \mathsf{ret}, \varepsilon, (1)\rangle \quad (\sigma_F = 0) \\
\vdash_{\mathsf{red}} \langle G\,(x-1)\,(H\,x) : \mathsf{ret} : \mathsf{ret}, \varepsilon, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle x - 1 : G[H\,x] : \mathsf{ret} : \mathsf{ret}, \varepsilon, (x) \cdot (1) : (1)\rangle \quad (\sigma_G = 1) \\
\vdash_{\mathsf{dec}} \langle x : 1 : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, \varepsilon, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle [x] : 1 : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, \varepsilon, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle x : \mathsf{ret} : 1 : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, \varepsilon, (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle [x] : \mathsf{ret} : 1 : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, \varepsilon, (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle \mathsf{ret} : 1 : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, 1, (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle 1 : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, 1, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle [1] : [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, 1, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle [-] : G[H\,x] : \mathsf{ret} : \mathsf{ret}, 1 : 1, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle G[H\,x] : \mathsf{ret} : \mathsf{ret}, 0, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle \mathsf{cond}\,(x = 0)\,x\,((G\,(x-1)\,y) + y) : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \varepsilon, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle x = 0 : \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \varepsilon, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle x : 0 : [=] : \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \varepsilon, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle [x] : 0 : [=] : \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \varepsilon, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle 0 : [=] : \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, 0, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle [0] : [=] : \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, 0, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle [=] : \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, 0 : 0, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle \mathsf{cond}[x, (G\,(x-1)\,y) + y] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \mathsf{true}, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle x : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \varepsilon, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{dec}} \langle [x] : \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, \varepsilon, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle \mathsf{ret} : \mathsf{ret} : \mathsf{ret}, 0, (0, H\,x) \cdot (x) \cdot (1) : (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle \mathsf{ret} : \mathsf{ret}, 0, (x) \cdot (1) : (1)\rangle \\
\vdash_{\mathsf{red}} \langle \mathsf{ret}, 0, (1)\rangle \\
\vdash_{\mathsf{red}} \langle \varepsilon, 0, \varepsilon\rangle
\end{array}
$$

Note that the environment stack, which is actually a nested stack of environments, could be implemented more efficiently by employing pointers to access the parameter values of previous (active) function calls.

Now we want to verify that the interpreter semantics in fact coincides with the reduction semantics. A closer look at the behaviour of our interpreter reveals that it simulates l–reductions so that those states actually occurring in a computation have a particular structure.

Definition 53 (L–environments): *The S^*–sorted set*

$$\mathbf{lEnv} \subseteq \mathbf{Env}$$

*of **l–environments** is inductively defined as follows.*

(i) *If* $a_i \in A^{s_i}$ *for* $1 \leq i \leq n$*, then* $(a_1, \ldots, a_n) \in \mathbf{lEnv}^{s_1 \cdots s_n}$.

(ii) *If* $e \in \mathbf{lEnv}^w$*,* $a_i \in A^{s_i}$ *for* $1 \leq i \leq n$ *or* $n = 0$*, and if* $t_i \in \mathbf{Dec}^{(w,s_i)} := T_{\Sigma[\mathbb{F}_\rho]}(\mathbb{X}_w)^{s_i}$ *for* $n{+}1 \leq i \leq m$*, then* $(a_1, \ldots, a_n, t_{n+1}, \ldots, t_m) \cdot e \in \mathbf{lEnv}^{s_1 \cdots s_m}$.

Due to these type restrictions, every l–environment yields a sequence of reduction terms which is obtained by iterated substitution.

Definition 54 (Reduction terms of l–environments): *The mapping*

$$\mathsf{rt} : \mathbf{lEnv} \to \mathbf{T}(R, \mathfrak{A})^*$$

is defined by

$$\mathsf{rt}((a_1, \ldots, a_n)) := (a_1, \ldots, a_n) \text{ and}$$
$$\mathsf{rt}((a_1, \ldots, a_n, t_{n+1}, \ldots, t_m) \cdot e) := (a_1, \ldots, a_n, t_{n+1}[\mathsf{rt}(e)], \ldots, t_m[\mathsf{rt}(e)]).$$

We see that rt preserves types such that the substitutions $t_i[\mathsf{rt}(e)]$, shorthand for $t_i[x^w/\mathsf{rt}(e)]$, are well defined.

Definition 55 (L–environment stack values): *A stack value* $es = e_1 : \ldots : e_n \in \mathbf{ES}$ *is called an* ***l–environment stack value*** *if it has the following properties:*

(i) $e_i \in \mathbf{lEnv}$ *for every* $1 \leq i \leq n$*,*

(ii) e_i *is compatible with* e_{i+1} *for every* $1 \leq i < n$*, i.e., we have either* $e_i = \bar{z} \cdot e_{i+1}$ *or* $e_{i+1} = \bar{z} \cdot e_i$*, for some* $\bar{z} \in Z^*$*.*

Definition 56 (L–states): *The* S*–sorted set* $\mathbf{lSt} \subseteq \mathbf{IntSt}$ *of* ***l–states****, together with their associated* ***reduction terms*** $\mathsf{rt} : \mathbf{lSt} \to \mathbf{T}(R, \mathfrak{A})$*, is inductively given as follows.*

(i) $st := \langle \varepsilon, a, \varepsilon \rangle \in \mathbf{lSt}^s$ *for every* $a \in A^s$*, and* $\mathsf{rt}(st) := a$*,*

(ii) $st := \langle \mathsf{ret}, a, e \rangle \in \mathbf{lSt}^s$ *for every* $a \in A^s$*,* $e \in \mathbf{lEnv}$*, and* $\mathsf{rt}(st) := a$*,*

(iii) $st := \langle [x_i^s] : \mathsf{ret}, \varepsilon, e \rangle \in \mathbf{lSt}^s$ *for every* $x_i^s \in \mathbb{X}$*,* $e \in \mathbf{lEnv}^{s_1 \cdots s_n}$ *with* $s_i = s$*, and* $\mathsf{rt}(st) := u_i$ *where* $\mathsf{rt}(e) = (u_1, \ldots, u_n)$*,*

(iv) $st := \langle [f] : \mathsf{ret}, a_1 : \ldots : a_n, e \rangle \in \mathbf{lSt}^s$ *for every* $f \in F^{(w,s)}_{\mathsf{base}}$*,* $(a_1, \ldots, a_n) \in A^w$*,* $e \in \mathbf{lEnv}$*, and* $\mathsf{rt}(st) := f a_1 \ldots a_n$*,*

(v) $st := \langle F_j^{(w,s)}[t_{\sigma_j+1}, \ldots, t_{n_j}] : \mathsf{ret}, a_1 : \ldots : a_{\sigma_j}, e \rangle \in \mathbf{lSt}^s$ *if* $F_j^{(w,s)}[t_{\sigma_j+1}, \ldots, t_{n_j}] \in \mathbf{Red}$*,* $w = s_1 \ldots s_{n_j}$*,* $e \in \mathbf{lEnv}^v$*,* $a_i \in A^{s_i}$*,* $t_i \in \mathbf{Dec}^{(v,s_i)}$*, and* $\mathsf{rt}(st) := F_j a_1 \ldots a_{\sigma_j} t_{\sigma_j+1}[\mathsf{rt}(e)] \ldots t_{n_j}[\mathsf{rt}(e)]$*,*

(vi) $st := \langle \mathsf{cond}_s[t_1, t_2] : \mathsf{ret}, \mathsf{true}, e \rangle \in \mathbf{lSt}^s$ *if* $\mathsf{cond}_s[t_1, t_2] \in \mathbf{Red}$*,* $t_1, t_2 \in \mathbf{Dec}^{(w,s)}$*,* $e \in \mathbf{lEnv}^w$*, and* $\mathsf{rt}(st) := \mathsf{cond}_s\, \mathsf{true}\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)]$*,*

(vii) $st := \langle \mathsf{cond}_s[t_1, t_2] : \mathsf{ret}, \mathsf{false}, e \rangle \in \mathbf{lSt}^s$ *if* $\mathsf{cond}_s[t_1, t_2] \in \mathbf{Red}$*,* $t_1, t_2 \in \mathbf{Dec}^{(w,s)}$*,* $e \in \mathbf{lEnv}^w$*, and* $\mathsf{rt}(st) := \mathsf{cond}_s\, \mathsf{false}\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)]$.

The next four rules specify the inductive closure. Under the induction hypothesis that $st = \langle ps : \mathsf{ret}, ds, es \rangle \in \mathbf{lSt}^s$ *with* $ps \neq \varepsilon$ *and* $\mathsf{rt}(st) = u \in \mathbf{T}(R, \mathfrak{A})^s$*, also the following states are l–states:*

(viii) $st := \langle ps : t_{k+1} : \ldots : t_n : [f] : \mathsf{ret}, a_1 : \ldots : a_{k-1} : ds, es \rangle \in \mathbf{lSt}^{s'}$ *for each* $f \in F^{(w,s')}_{\mathsf{base}}$*,* $w = s_1 \ldots, s_n$*,* $a_i \in A^{s_i}$*,* $t_i \in \mathbf{Dec}^{(v,s_i)}$*,* $s = s_k$*,* $es = e : es'$*,* $e \in \mathbf{lEnv}^v$*, and* $\mathsf{rt}(st) := f a_1 \ldots a_{k-1} u\, t_{k+1}[\mathsf{rt}(e)] \ldots t_n[\mathsf{rt}(e)]$*,*

(ix) $st := \langle ps:\mathsf{cond}_{s'}[t_1,t_2]:\mathsf{ret}, ds, es\rangle \in \mathbf{lSt}^{s'}$ *if* $s = \mathsf{bool}$, $cond_{s'}[t_1,t_2] \in \mathbf{Red}$, $t_1, t_2 \in \mathbf{Dec}^{(w,s')}$, $es = e:es'$, $e \in \mathbf{lEnv}^w$, *and* $\mathsf{rt}(st) := \mathsf{cond}_{s'}\, u\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)]$,

(x) $st := \langle ps:t_{k+1}:\ldots:t_{\sigma_j} : F_j^{(w,s')}[t_{\sigma_j+1},\ldots,t_{n_j}]:\mathsf{ret}, a_1:\ldots:a_{k-1}:ds, es\rangle \in \mathbf{lSt}^{s'}$ *if* $F_j^{(w,s')}[t_{\sigma_j+1},\ldots,t_{n_j}] \in \mathbf{Red}$, $w = s_1\ldots s_{n_j}$, $a_i \in A^{s_i}$, $t_i \in \mathbf{Dec}^{(v,s_i)}$, $s = s_k$, $es = e:es'$, $e \in \mathbf{lEnv}^v$, *and* $\mathsf{rt}(st) := F_j a_1 \ldots a_{k-1} u\, t_{k+1}[\mathsf{rt}(e)] \ldots t_{n_j}[\mathsf{rt}(e)]$,

(xi) $st := \langle ps:\mathsf{ret}:\mathsf{ret}, ds, es:e\rangle \in \mathbf{lSt}^s$ *if* $e \in \mathbf{lEnv}$, *and* $\mathsf{rt}(st) := u$.

Observe that each l–state is a reduction state as it is in decomposed form.

Lemma 57: *Let* $(w,s) \in S^* \times S$, $e \in \mathbf{lEnv}^w$, *and* $t \in \mathbf{Dec}^{(w,s)}$. *Then it holds for* $st = \langle t:\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{IntSt}$ *that* $\mathsf{dec}(st) = \langle \mathsf{dec}(t):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt}^s$ *and* $\mathsf{rt}(\mathsf{dec}(st)) = t[\mathsf{rt}(e)]$.

Proof: by induction on t.

(i) For $t = x_i \in \mathbb{X}$ the assertion follows from Definition 56 together with the observation that $x_i[\mathsf{rt}(e)] = u_i$ if $\mathsf{rt}(e) = (u_1,\ldots,u_n)$.

(ii) For $t = c \in C$ we similarly conclude that $\mathsf{dec}(st) \in \mathbf{lSt}$ and $\mathsf{rt}(st) = c = c[\mathsf{rt}(e)]$.

(iii) Let $t = f t_1 \ldots t_n \in \mathbf{Dec}$ where $n \geq 1$. Hence, $t_1 \in \mathbf{Dec}$, and by induction hypothesis, $\langle \mathsf{dec}(t_1):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt}$ and $\mathsf{rt}(\langle \mathsf{dec}(t_1):\mathsf{ret}, \varepsilon, e\rangle) = t_1[\mathsf{rt}(e)]$. Then Definition 56 implies that

$$\begin{aligned} \langle \mathsf{dec}(t_1):t_2:\ldots:t_n:[f]:\mathsf{ret}, \varepsilon, e\rangle &= \langle \mathsf{dec}(t):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt} \quad \text{and} \\ \mathsf{rt}(\mathsf{dec}(st)) &= f\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)] \ldots t_n[\mathsf{rt}(e)] \\ &= t[\mathsf{rt}(e)]. \end{aligned}$$

(iv) Let $t = \mathsf{cond}\, t_0 t_1 t_2 \in \mathbf{Dec}$. Then $t_0 \in \mathbf{Dec}$, and by induction $\langle \mathsf{dec}(t_0):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt}$ and $\mathsf{rt}(\langle \mathsf{dec}(t_0):\mathsf{ret}, \varepsilon, e\rangle) = t_0[\mathsf{rt}(e)]$. Again Definition 56 implies that

$$\begin{aligned} \langle \mathsf{dec}(t_0):\mathsf{cond}[t_1,t_2]:\mathsf{ret}, \varepsilon, e\rangle &= \langle \mathsf{dec}(t):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt} \quad \text{and} \\ \mathsf{rt}(\mathsf{dec}(st)) &= \mathsf{cond}\, t_0[\mathsf{rt}(e)]\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)] \\ &= t[\mathsf{rt}(e)]. \end{aligned}$$

(v) For $t = F_j t_1 \ldots t_{n_j} \in \mathbf{Dec}$ where $\sigma_j \geq 1$ and, inductively, $\langle \mathsf{dec}(t_1):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt}$ and $\mathsf{rt}(\langle \mathsf{dec}(t_1):\mathsf{ret}, \varepsilon, e\rangle) = t_1[\mathsf{rt}(e)]$ we conclude from Definition 56 that

$$\begin{aligned} \langle \mathsf{dec}(t_1):t_2:\ldots:t_{\sigma_j}:F_j[t_{\sigma_j+1},\ldots,t_{n_j}]:\mathsf{ret}, \varepsilon, e\rangle &= \langle \mathsf{dec}(t):\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt} \quad \text{and} \\ \mathsf{rt}(\mathsf{dec}(st)) &= F_j\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)] \ldots t_{n_j}[\mathsf{rt}(e)] \\ &= t[\mathsf{rt}(e)]. \end{aligned}$$

(vi) For $t = F_j t_1 \ldots t_{n_j} \in \mathbf{Dec}$ where $\sigma_j = 0$ we directly see from Definition 56 that

$$\begin{aligned} \langle \mathsf{dec}(t):\mathsf{ret}, \varepsilon, e\rangle &= \langle F_j[t_1,\ldots,t_{n_j}]:\mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt} \quad \text{and} \\ \mathsf{rt}(\mathsf{dec}(st)) &= t[\mathsf{rt}(e)]. \end{aligned}$$

□

Now we are well prepared to prove that our interpreter implements the left–reduction strategy. For this purpose we start from the obvious correspondence between initial states, and then we show that this correspondence between interpreter states and left–reduction terms is preserved during computation.

Lemma 58 (Correspondence of initial states): *Let* $(R, \mathfrak{A}) \in \mathbf{Rfd}_{\Sigma}^{(w,s)}$ *and* $\bar{a} = (a_1, \ldots, a_n) \in A^w$. *Then it holds for the initial state* $st_{\bar{a}} := \langle F_1 x_1 \ldots x_n : \mathsf{ret}, \varepsilon, \bar{a}\rangle$ *that* $\mathsf{dec}(st_{\bar{a}}) \in \mathbf{lSt}$ *and* $\mathsf{rt}(\mathsf{dec}(st_{\bar{a}})) = F_1 a_1 \ldots a_n$.

Proof: The claim follows directly from the previous lemma. □

Theorem 59: *For each interpreter state* $st = \langle ps, ds, es\rangle \in \mathbf{lSt}$ *with* $ps \neq \varepsilon$ *there is* $st' \in \mathbf{IntSt}$ *such that*

$$st \vdash_{\mathsf{red}} st' \vdash^*_{\mathsf{dec}} \mathsf{dec}(st') \in \mathbf{lSt}$$

and

$$\mathsf{rt}(st) \Rightarrow^n_l \mathsf{rt}(\mathsf{dec}(st')) \textit{ for some } n \in \{0, 1\}.$$

Proof: by induction on st according to Definition 56.

(ii) For $st = \langle \mathsf{ret}, a, e\rangle$ we have $st \vdash_{\mathsf{red}} \langle \varepsilon, a, \varepsilon\rangle \in \mathbf{lSt}$ and $\mathsf{rt}(st) = a = \mathsf{rt}(\langle \varepsilon, a, \varepsilon\rangle)$ so that the claim holds with $n = 0$.

(iii) For $st = \langle [x_i] : \mathsf{ret}, \varepsilon, (z_1, \ldots, z_k) \cdot e\rangle$, the following cases are possible:

(a) $z_i \in A$: here $st \vdash_{\mathsf{red}} \langle \mathsf{ret}, z_i, (z_1, \ldots, z_k) \cdot e\rangle =: st'$, and hence $\mathsf{dec}(st') = st'$ and $\mathsf{rt}(st) = z_i = \mathsf{rt}(\mathsf{dec}(st'))$.

(b) $z_i \in \mathbf{Dec}$: we have $st \vdash_{\mathsf{red}} \langle z_i : \mathsf{ret} : \mathsf{ret}, \varepsilon, e : (z_1, \ldots, z_k) \cdot e\rangle =: st'$. Here $\mathsf{dec}(st') \in \mathbf{lSt}$ holds since Lemma 57 implies that $\langle \mathsf{dec}(z_i) : \mathsf{ret}, \varepsilon, e\rangle \in \mathbf{lSt}$, and from Definition 56 we see that also $\mathsf{dec}(st') = \langle \mathsf{dec}(z_i) : \mathsf{ret} : \mathsf{ret}, \varepsilon, e : (z_1, \ldots, z_k) \cdot e\rangle \in \mathbf{lSt}$. For the corresponding reduction terms we conclude $\mathsf{rt}(st) = z_i[\mathsf{rt}(e)]$, and, again by Lemma 57, $\mathsf{rt}(\mathsf{dec}(st')) = \mathsf{rt}(\langle \mathsf{dec}(z_i) : \mathsf{ret}, \varepsilon, e\rangle) = z_i[\mathsf{rt}(e)]$.

(iv) If $st = \langle [f] : \mathsf{ret}, a_1 : \ldots : a_n, e\rangle \in \mathbf{lSt}$ and $a = f_{\mathfrak{A}}(a_1, \ldots, a_n)$, then $st \vdash_{\mathsf{red}} \langle \mathsf{ret}, a, e\rangle \in \mathbf{lSt}$ and $\mathsf{rt}(st) = f a_1 \ldots a_n \Rightarrow_l a = \mathsf{rt}(\langle \mathsf{ret}, a, e\rangle)$.

(v) If $st = \langle F_j[t_{\sigma_j+1}, \ldots, t_{n_j}] : \mathsf{ret}, a_1 : \ldots : a_{\sigma_j}, e\rangle \in \mathbf{lSt}$, then

$$st \vdash_{\mathsf{red}} \langle dt_j : \mathsf{ret} : \mathsf{ret}, \varepsilon, (a_1, \ldots, a_{\sigma_j}, t_{\sigma_j+1}, \ldots, t_{n_j}) \cdot e : e\rangle =: st'.$$

Here, $\mathsf{dec}(st') \in \mathbf{lSt}$ follows as in case (iii)(b), and

$$\begin{aligned}
\mathsf{rt}(st) &= F_j a_1 \ldots a_{\sigma_j} t_{\sigma_j+1}[\mathsf{rt}(e)] \ldots t_{n_j}[\mathsf{rt}(e)] \\
&\Rightarrow_l dt_j[(a_1, \ldots, a_{\sigma_j}, t_{\sigma_j+1}[\mathsf{rt}(e)], \ldots, t_{n_j}[\mathsf{rt}(e)])] \\
&= dt_j[\mathsf{rt}((a_1, \ldots, a_{\sigma_j}, t_{\sigma_j+1}, \ldots, t_{n_j}) \cdot e)] \\
&= \mathsf{rt}(\mathsf{dec}(st')).
\end{aligned}$$

(vi) If $st = \langle \mathsf{cond}[t_1, t_2] : \mathsf{ret}, \mathsf{true}, e\rangle \in \mathbf{lSt}$, then $st \vdash_{\mathsf{red}} \langle t_1 : \mathsf{ret}, \varepsilon, e\rangle =: st'$. Accordingly, $\mathsf{dec}(st') \in \mathbf{lSt}$, and $\mathsf{rt}(st) = \mathsf{cond}\,\mathsf{true}\, t_1[\mathsf{rt}(e)]\, t_2[\mathsf{rt}(e)] \Rightarrow_l t_1[\mathsf{rt}(e)] = \mathsf{rt}(\mathsf{dec}(st'))$.

(vii) The case of false can be dealt with analogously.

The remaining cases concern inductive steps. We first observe a general property of reductions: if

$$\langle ps : \mathsf{ret}, ds, es\rangle \vdash_{\mathsf{red}} \langle ps' : \mathsf{ret}, ds', es'\rangle,$$

then

$$\langle ps : ps_1 : \mathsf{ret}, ds_1 : ds, es : es_1\rangle \vdash_{\mathsf{red}} \langle ps' : ps_1 : \mathsf{ret}, ds_1 : ds', es' : es_1\rangle$$

for every possible extension of the interpreter state. This holds since reduction steps only modify a limited amount of upper stack elements whereas the extensions modify stacks at the other end. In addition, the program stack must contain a non–empty ps such that the ret entry remains unchanged.

Now we assume inductively that $st = \langle ps : \mathsf{ret}, ds, es\rangle \vdash_{\mathsf{red}} st' = \langle ps' : \mathsf{ret}, ds', es'\rangle$ with $st, \mathsf{dec}(st') \in \mathbf{lSt}$ and $\mathsf{rt}(st) \Rightarrow_l^n \mathsf{rt}(\mathsf{dec}(st'))$ for some $n \in \{0, 1\}$. In each of the following cases it follows from the property stated above that the extended state, $\widehat{st}$, leads again to an l–state:

$$\widehat{st} \vdash_{\mathsf{red}} \widehat{st}' \vdash^*_{\mathsf{dec}} \mathsf{dec}(\widehat{st}') \in \mathbf{lSt}.$$

More precisely, the reduction of $\widehat{st}$ extends the reduction of st in the same way as $\widehat{st}$ extends st: $\widehat{st}' = \widehat{st'}$; furthermore, $\mathsf{dec}(\widehat{st}') = \mathsf{dec}(\widehat{st'}) = \widehat{\mathsf{dec}(st')} \in \mathbf{lSt}$. Altogether we see that our interpreter in fact preserves the l–property of states.

We still have to verify that $\mathsf{rt}(\widehat{st}) \Rightarrow_l^n \mathsf{rt}(\mathsf{dec}(\widehat{st}'))$. Here we have to treat the four possible extensions separately.

(viii) For $\widehat{st} = \langle ps : t_{k+1} : \ldots : t_n : [f] : \mathsf{ret}, a_1 : \ldots : a_{k-1} : ds, es\rangle$ and $\mathsf{rt}(\widehat{st}) = f a_1 \ldots a_{k-1}\, \mathsf{rt}(st)\, t_{k+1}[\mathsf{rt}(e)] \ldots t_n[\mathsf{rt}(e)]$ where $es = e : es_1$, the induction hypothesis yields

$$\begin{aligned}
\mathsf{rt}(\widehat{st}) &\Rightarrow_l^n f a_1 \ldots a_{k-1}\, \mathsf{rt}(\mathsf{dec}(st'))\, t_{k+1}[\mathsf{rt}(e)] \ldots t_n[\mathsf{rt}(e)] \\
&= \mathsf{rt}(\langle \mathsf{dec}(ps') : t_{k+1} : \ldots : t_n : [f] : \mathsf{ret}, a_1 : \ldots : a_{k-1} : ds', es'\rangle) \\
&= \mathsf{rt}(\mathsf{dec}(\langle ps' : t_{k+1} : \ldots : t_n : [f] : \mathsf{ret}, a_1 : \ldots : a_{k-1} : ds', es'\rangle)) \\
&= \mathsf{rt}(\mathsf{dec}(\widehat{st'})) \\
&= \mathsf{rt}(\mathsf{dec}(\widehat{st}')).
\end{aligned}$$

In the remaining cases the claim follows in the same way. $\square$

These results show that a computation can be viewed as a sequence of **l–transitions**, where the **l–transition relation** $\vdash_l \subseteq \mathbf{lSt} \times \mathbf{lSt}$ is defined by

$$st \vdash_l st' \quad \text{if} \quad st \vdash_{\mathsf{red}} st'' \vdash^*_{\mathsf{dec}} st' \quad \text{for some } st'' \in \mathbf{IntSt}.$$

Note that if st' and st'' exist, they are uniquely determined by st.

Corollary 60: *For* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma^{(w,s)}, \bar{a} \in A^w$, $b \in A^s$ *and* $st_0 := \mathsf{dec}(st_{\bar{a}})$ *we have*

(i) $\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = b$ *iff* $st_0 \vdash_l^* \langle \varepsilon, b, \varepsilon\rangle$ *and*
(ii) $\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = \perp$ *iff there is an infinite computation* $(st_i \vdash_l st_{i+1} \mid i \in \mathbb{N})$.

Proof: This is an immediate consequence of our previous results taking into account that $\langle \varepsilon, ds, es\rangle \in \mathbf{lSt}$ implies that $ds \in A$ and $es = \varepsilon$. $\square$

Theorem 61 (Equivalence of interpreter and l–reduction semantics): *It holds for every $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ that*

$$\mathsf{Int}[\![R]\!]_{\mathfrak{A}} = \mathsf{lRd}[\![R]\!]_{\mathfrak{A}}.$$

Proof: If $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma^{(w,s)}$, then both semantic functions are of type $A^w \to A^s_\perp$. Hence, it suffices to verify that

(i) $\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = b$ implies $\mathsf{lRd}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = b$ and
(ii) $\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = \perp$ implies $\mathsf{lRd}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = \perp$

for each $\bar{a} \in A^w$ and $b \in A^s$.

Proof of (i): If $\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = b$, there is a sequence of l–transitions $(st_i \vdash_l st_{i+1} \mid i = 0, \dots, q-1)$ such that $st_0 = \mathsf{dec}(st_{\bar{a}})$ and $st_q = \langle \varepsilon, b, \varepsilon \rangle$. For the corresponding reduction terms $\mathsf{rt}(st_i)$ we conclude that $\mathsf{rt}(st_0) = F_1\bar{a}$, $\mathsf{rt}(st_q) = b$ and $\mathsf{rt}(st_i) \Rightarrow_l^n \mathsf{rt}(st_{i+1})$ for some $n \in \{0, 1\}$ and $i = 0, \dots, q-1$. Hence, $F_1\bar{a} \Rightarrow_l^* b$ and thereby $\mathsf{lRd}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = b$.

Proof of (ii): This case is more subtle in so far as we have to distinguish l–transitions corresponding to proper l–reduction steps ($n = 1$) from those with equal reduction terms ($n = 0$). Therefore we define for an l–transition $st \vdash_l st'$ that

$$st \vdash_{l0} st' \quad \text{if} \quad \mathsf{rt}(st) = \mathsf{rt}(st') \quad \text{and}$$
$$st \vdash_{l1} st' \quad \text{if} \quad \mathsf{rt}(st) \Rightarrow_l \mathsf{rt}(st').$$

Now, if $\mathsf{Int}[\![R]\!]_{\mathfrak{A}}(\bar{a}) = \perp$, there is an infinite computation $(st_i \vdash_l st_{i+1} \mid i \in \mathbb{N})$ starting from $st_0 = \mathsf{dec}(st_{\bar{a}})$. It remains to show that the corresponding l–reduction sequence is infinite, too.

For that purpose we prove that an $l0$–transition $st = \langle ps, ds, es \rangle \vdash_{l0} st' = \langle ps', ds', es' \rangle$ has the following property. It holds that either

(a) st' is final, i.e., $st' = \langle \varepsilon, b, \varepsilon \rangle$ for some $b \in A$, or
(b) es is extended by a shortened environment, i.e., $es = (z_1, \dots, z_k) \cdot e : es_0$ and $es' = e : es$, or
(c) es is preserved and ps is shortened, i.e., $es' = es$ and $ps = p : ps'$.

As a consequence there cannot be an infinite sequence of $l0$–transitions so that in fact our l–reduction sequence must be infinite.

We prove this $l0$–property by induction on $st \in \mathbf{lSt}$:

(i) $st = \langle \varepsilon, a, \varepsilon \rangle$ does not allow any $l0$–transition.
(ii) $st = \langle \mathsf{ret}, a, e \rangle \in \mathbf{lSt}$ yields $st \vdash_{l0} \langle \varepsilon, a, \varepsilon \rangle$ with property (a).
(iii) $st = \langle [x_i] : \mathsf{ret}, \varepsilon, (z_1, \dots, z_m) \cdot e \rangle \in \mathbf{lSt}$:
if $z_i \in A$, then $st \vdash_{l0} \langle \mathsf{ret}, z_i, (z_1, \dots, z_m) \cdot e \rangle$ with property (c);
if $z_i \in T$, then $st \vdash_{l0} \langle \mathsf{dec}(z_i) : \mathsf{ret} : \mathsf{ret}, \varepsilon, e : (z_1, \dots, z_m) \cdot e \rangle$ with property (b);
(iv) - (vii) do not allow any $l0$–transition.

Now we assume as induction hypothesis that $st = \langle ps : \mathsf{ret}, ds, es \rangle \in \mathbf{lSt}$ with $ps \neq \varepsilon$ and that $st \vdash_{l0} st'$ has the $l0$–property.

(viii) Let $st_1 = \langle ps : t_{k+1} : \dots : t_n : [f] : \mathsf{ret}, a_1 : \dots : a_{k-1} : ds, es \rangle$.
If $st_1 \vdash_{l0} st_1'$, it follows that $st = \langle ps : \mathsf{ret}, ds, es \rangle \vdash_{l0} st' = \langle ps' : \mathsf{ret}, ds', es' \rangle$ for suitable ps', ds', and es', and in addition that $st_1' = \langle ps' : t_{k+1} : \dots : t_n : [f] : \mathsf{ret}, a_1 : \dots : a_{k-1} : ds', es' \rangle$. Hence, the $l0$–property is preserved.

The remaining cases can be treated analogously. □

5 A Compiler for Recursive Function Definitions

In a final step we now transform the interpreter into a compiler. For this purpose we exploit the fact that only subterms of righthand sides dt_j and of the calling term $F_1 x_1 \dots x_{n_1}$ may occur as decomposition symbols in a computation. Their reduction symbols are taken as machine commands. All symbols will be replaced by addresses in order to eliminate the implicit control of reduction steps by means of term decomposition and to use instead explicit control by jumps. Only connecting the ret command requires a dynamic control through a return stack. Hence, the program stack will change into a program counter together with a return stack.

First we modify the interpreter replacing symbols by appropriate addresses. Then we show how these addresses permit an explicit control of reduction steps. Thereafter an abstract stack machine together with a compiler emerge as a natural consequence.

Definition 62 (Addresses and their symbols): *Let* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ *and* $R = (F_j^{\tau_j} = \lambda^{\sigma_j} x^{w_j}.dt_j \mid 1 \leq j \leq r)$. *The set* $\mathbf{Adr}_R \subseteq \mathbb{N}^*$ *of* ***addresses*** *of* R *and their decomposition symbols, given by* $\mathsf{decsym} : \mathbf{Adr}_R \to \mathbf{Dec}$, *are defined by*

- $j \in \mathbf{Adr}_R$ *and* $\mathsf{decsym}(j) := dt_j$ *for each* $j \in \{1, \dots, r\}$,
- $0 \in \mathbf{Adr}_R$ *and* $\mathsf{decsym}(0) := F_1 x^{w_1}$, *and*
- *if* $\alpha \in \mathbf{Adr}_R$, $\mathsf{decsym}(\alpha) = \varphi t_1 \dots t_m$, *and* $1 \leq i \leq m$, *then also* $\alpha.i \in \mathbf{Adr}_R$ *and* $\mathsf{decsym}(\alpha.i) = t_i$.

In addition, let $\mathbf{Adr}_R^{\mathsf{ret}} := \mathbf{Adr}_R \cup \{r+1\}$.
Each address $\alpha \in \mathbf{Adr}_R^{\mathsf{ret}}$ *also determines a reduction symbol:*

- $\mathsf{redsym}(\alpha) := \mathsf{redsym}(\mathsf{decsym}(\alpha))$ *for* $\alpha \neq r+1$, *and*
- $\mathsf{redsym}(r+1) := \mathsf{ret}$.

Note that $\mathbf{Adr}_R$ and therefore $\mathsf{decsym}(\mathbf{Adr}_R)$ and $\mathsf{redsym}(\mathbf{Adr}_R^{\mathsf{ret}})$ are finite sets. Since only their elements may occur in an actual computation, we can replace program and environment stack symbols by corresponding addresses. On the program stack we have to distinguish between decomposition and reduction addresses whereas on the environment stack an address always refers to a decomposition symbol.

The interpreter with addresses will be constructed with respect to $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$. In contrast, the abstract stack machine has to work for all recursive function definitions. Therefore we use the index R for denoting the sets of addresses and of environment stack values and drop this index later with the abstract stack machine.

Definition 63 (Interpreter with addresses): *The* ***address interpreter*** *of* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ *is defined by the set*

$$\mathbf{IntSt}^@ := \mathbf{PS}^@ \times \mathbf{DS} \times \mathbf{ES}_R^@$$

of ***address states*** *where*

$$\begin{aligned} \mathbf{PS}^@ &:= (\{(\alpha, d) \mid \alpha \in \mathbf{Adr}_R\} \cup \{(\alpha, r) \mid \alpha \in \mathbf{Adr}_R^{\mathsf{ret}}\})^*, \\ \mathbf{DS} &:= A^*, \text{ and} \\ \mathbf{ES}_R^@ &:= (\mathbf{Env}_R^@)^* \text{ where } \mathbf{Env}_R^@ := ((A \cup \mathbf{Adr}_R)^*)^*, \end{aligned}$$

by the corresponding transition relation

$$\vdash := \vdash_{\mathsf{dec}} \cup \vdash_{\mathsf{red}} \subseteq \mathbf{IntSt}^{@} \times \mathbf{IntSt}^{@}$$

which is given by

- $\langle (\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle (\alpha, r) : ps, ds, es\rangle$
  if $\mathsf{decsym}(\alpha) = x$,
- $\langle (\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle (\alpha.1, d) : \ldots : (\alpha.n, d) : (\alpha, r) : ps, ds, es\rangle$
  if $\mathsf{decsym}(\alpha) = f t_1 \ldots t_n$,
- $\langle (\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle (\alpha.1, d) : (\alpha, r) : ps, ds, es\rangle$
  if $\mathsf{decsym}(\alpha) = \mathsf{cond} t_0 t_1 t_2$,
- $\langle (\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle (\alpha.1, d) : \ldots : (\alpha.\sigma_j, d) : (\alpha, r) : ps, ds, es\rangle$
  if $\mathsf{decsym}(\alpha) = F_j t_1 \ldots t_{n_j}$,

- $\langle (\alpha, r) : ps, ds, \bar{z} \cdot e : es\rangle \vdash_{\mathsf{red}} \langle ps, ds : z_i, \bar{z} \cdot e : es\rangle$
  if $\mathsf{redsym}(\alpha) = [x_i]$, $\bar{z} = (z_1, \ldots, z_k)$, $1 \leq i \leq k$, *and* $z_i \in A$,
- $\langle (\alpha, r) : ps, ds, \bar{z} \cdot e : es\rangle \vdash_{\mathsf{red}} \langle (z_i, d) : (r + 1, r) : ps, ds, e : \bar{z} \cdot e : es\rangle$
  if $\mathsf{redsym}(\alpha) = [x_i]$, $\bar{z} = (z_1, \ldots, z_k)$, $1 \leq i \leq k$, *and* $z_i \in \mathbf{Adr}_R$,
- $\langle (\alpha, r) : ps, ds : a_1 : \ldots : a_n, es\rangle \vdash_{\mathsf{red}} \langle ps, ds : f_{\mathfrak{A}}(a_1, \ldots, a_n), es\rangle$
  if $\mathsf{redsym}(\alpha) = [f]$,
- $\langle (\alpha, r) : ps, ds : \mathsf{true}, es\rangle \vdash_{\mathsf{red}} \langle (\alpha.2, d) : ps, ds, es\rangle$
  if $\mathsf{redsym}(\alpha) = \mathsf{cond}[t_1, t_2]$,
- $\langle (\alpha, r) : ps, ds : \mathsf{false}, es\rangle \vdash_{\mathsf{red}} \langle (\alpha.3, d) : ps, ds, es\rangle$
  if $\mathsf{redsym}(\alpha) = \mathsf{cond}[t_1, t_2]$,
- $\langle (\alpha, r) : ps, ds : a_1 : \ldots : a_{\sigma_j}, e : es\rangle \vdash_{\mathsf{red}}$
  $\langle (j, d) : (r + 1, r) : ps, ds, (a_1, \ldots, a_{\sigma_j}, \alpha.(\sigma_j + 1), \ldots, \alpha.n_j) \cdot e : e : es\rangle$
  if $\mathsf{redsym}(\alpha) = F_j[t_{\sigma_j+1}, \ldots, t_{n_j}]$,
- $\langle (r + 1, r) : ps, ds, e : es\rangle \vdash_{\mathsf{red}} \langle ps, ds, es\rangle$,

*and by the **address interpreter semantics***

$$\mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}} : A^w \to A^s_\perp$$

where

$$\mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}}(a_1, \ldots, a_{n_1}) := \begin{cases} b & \text{if } \langle (0, d) : (r + 1, r), \varepsilon, (a_1, \ldots, a_{n_1})\rangle \vdash^* \langle \varepsilon, b, \varepsilon\rangle, \\ \perp^s & \text{if no such } b \in A^s \text{ exists.} \end{cases}$$

From this address abstraction it follows directly that the semantics remains unchanged.

Corollary 64: *For* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ *we have*

$$\mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}} = \mathsf{Int}[\![R]\!]_{\mathfrak{A}}.$$

Proof: Replacing addresses by corresponding decomposition and reduction symbols an address interpreter computation turns into an equivalent interpreter computation. □

Addresses of a program stack value occurring in an actual computation are connected in a particular way. We describe this connection by two functions, first and follow. $\mathsf{first}(\alpha)$ gives for $\mathsf{decsym}(\alpha)$ the start address of the corresponding computation, i.e., the address of a subterm whose reduction symbol causes the first reduction step. It is determined by decomposition of $\mathsf{decsym}(\alpha)$. $\mathsf{follow}(\alpha)$ addresses the reduction symbol where the computation continues after evaluating $\mathsf{decsym}(\alpha)$.

Definition 65: $\mathsf{first} : \mathbf{Adr}_R^{\mathsf{ret}} \to \mathbf{Adr}_R^{\mathsf{ret}}$ *and* $\mathsf{follow} : \mathbf{Adr}_R \to \mathbf{Adr}_R^{\mathsf{ret}}$ *are defined by*

$$\mathsf{first}(\alpha) := \begin{cases} \alpha & \text{if } \mathsf{decsym}(\alpha) \in \mathbb{X} \cup C \\ & \text{or } \mathsf{decsym}(\alpha) = F_j\, t_1 \dots t_{n_j}, \sigma_j = 0 \\ & \text{or } \alpha = r+1, \\ \mathsf{first}(\alpha.1) & \text{if } \mathsf{decsym}(\alpha) = f\, t_1 \dots t_n, n \geq 1 \\ & \text{or } \mathsf{decsym}(\alpha) = \mathsf{cond}\, t_0\, t_1\, t_2 \\ & \text{or } \mathsf{decsym}(\alpha) = F_j\, t_1 \dots t_{n_j}, \sigma_j \geq 1, \end{cases}$$

$$\mathsf{follow}(j) := \; r+1 \quad \text{if } 0 \leq j \leq r,$$

$$\mathsf{follow}(\alpha.i) := \begin{cases} \mathsf{first}(\alpha.(i+1)) & \text{if } \mathsf{decsym}(\alpha) = f\, t_1 \dots t_n,\ 1 \leq i < n \\ & \text{or } \mathsf{decsym}(\alpha) = F_j\, t_1 \dots t_{n_j},\ 1 \leq i < \sigma_j, \\ \alpha & \text{if } \mathsf{decsym}(\alpha) = f\, t_1 \dots t_n,\ i = n \\ & \text{or } \mathsf{decsym}(\alpha) = F_j\, t_1 \dots t_{n_j},\ i = \sigma_j \\ & \text{or } \mathsf{decsym}(\alpha) = \mathsf{cond}\, t_0\, t_1\, t_2,\ i = 1, \\ r+1 & \text{if } \mathsf{decsym}(\alpha) = F_j\, t_1 \dots t_{n_j},\ \sigma_j < i \leq n_j, \\ \mathsf{follow}(\alpha) & \text{if } \mathsf{decsym}(\alpha) = \mathsf{cond}\, t_0\, t_1\, t_2,\ i \in \{2,3\}. \end{cases}$$

Definition 66: ***Connected*** *program stack values* $ps \in \mathbf{PS}^{@}$ *are defined by the following induction:*

- ε *is connected,*
- *if* ps *is connected, then* $(r+1, r) : ps$ *is connected,*
- *if* $(\alpha, d) : ps$ *is connected and* $\beta \in \mathbf{Adr}_R$ *such that* $\mathsf{follow}(\beta) = \mathsf{first}(\alpha)$, *then* $(\beta, \nu) : (\alpha, d) : ps$ *is connected for* $\nu \in \{d, r\}$,
- *if* $(\alpha, r) : ps$ *is connected and* $\beta \in \mathbf{Adr}_R$ *such that* $\mathsf{follow}(\beta) = \alpha$, *then* $(\beta, \nu) : (\alpha, r) : ps$ *is connected for* $\nu \in \{d, r\}$.

Definition 67: *An address state is called* ***reachable*** *if it occurs in an actual computation, i.e.,*

- $\langle (0, d) : (r+1, r), \varepsilon, (a_1, \dots, a_{n_1}) \rangle$ *is reachable for all* $(a_1, \dots, a_{n_1}) \in A^{n_1}$,
- *if* $\langle ps, ds, es \rangle \in \mathbf{IntSt}^{@}$ *is reachable and* $\langle ps, ds, es \rangle \vdash \langle ps', ds', es' \rangle$, *then* $\langle ps', ds', es' \rangle$ *is reachable.*

Lemma 68: *Reachable address states have connected program stack values.*

Proof: (i) The initial program stack value $(0, d) : (r+1, r)$ is connected because $(r+1, r)$ is connected and $\mathsf{follow}(0) = r+1$.

(ii) Decomposition steps preserve the connectedness of program stack values. To show this, let $st := \langle(\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} st'$ and $(\alpha, d) : ps$ be connected. It follows that $ps = (\beta, x) : ps'$ and that the follow–condition holds for α and β. There are four types of decomposition steps given by $\mathsf{decsym}(\alpha)$.

- $\mathsf{decsym}(\alpha) = x$.
  It follows that $st' = \langle(\alpha, r) : ps, ds, es\rangle$. Since the follow–condition of the upper two stack elements remains unchanged, $(\alpha, r) : ps$ is connected, too.
- $\mathsf{decsym}(\alpha) = ft_1 \ldots t_n$.
  Hence, $st' = \langle(\alpha.1, d) : \ldots : (\alpha.n, d) : (\alpha, r) : ps, ds, es\rangle$. We conclude as in the previous case that $(\alpha, r) : ps$ is connected. Moreover, $\mathsf{follow}(\alpha.i) = \mathsf{first}(\alpha.(i + 1))$ for $i = 1, \ldots, n - 1$ and $\mathsf{follow}(\alpha.n) = \alpha$ which shows that the program stack value of st' is connected.
- $\mathsf{decsym}(\alpha) = \mathsf{cond}t_0t_1t_2$.
  Here, $st' = \langle(\alpha.1, d) : (\alpha, r) : ps, ds, es\rangle$. Its program stack value is connected because this holds for $(\alpha, r) : ps$ and $\mathsf{follow}(\alpha.1) = \alpha$.
- $\mathsf{decsym}(\alpha) = F_jt_1 \ldots t_{n_j}$.
  It follows that $st' = \langle(\alpha.1, d) : \ldots : (\alpha.\sigma_j, d) : (\alpha, r) : ps, ds, es\rangle$. Again, we easily check that addresses are connected appropriately.

(iii) Reduction steps also preserve the connectedness of program stack values. There are seven cases:

- $\langle(\alpha, r) : ps, ds, \bar{z} \cdot e : es\rangle \vdash_{\mathsf{red}} \langle ps, ds : z_i, \bar{z} \cdot e : es\rangle$ where $\mathsf{redsym}(\alpha) = [x_i]$, $\bar{z} = (z_1, \ldots, z_k)$, $1 \le i \le k$, and $z_i \in A$.
  Here, ps directly inherits connectedness from $(\alpha, r) : ps$.
- $\langle(\alpha, r) : ps, ds, \bar{z} \cdot e : es\rangle \vdash_{\mathsf{red}} \langle(z_i, d) : (r + 1, r) : ps, ds, e : \bar{z} \cdot e : es\rangle$ where $\mathsf{redsym}(\alpha) = [x_i]$, $\bar{z} = (z_1, \ldots, z_k)$, $1 \le i \le k$, and $z_i \in \mathbf{Adr}_R$.
  In this case, $(z_i, d) : (r+1, r) : ps$ is connected because this holds for ps and therefore for $(r + 1, r) : ps$, and because for $z_i \in \mathbf{Adr}_R$ being the address of a delayed function call argument we have $\mathsf{follow}(z_i) = r + 1$.
- $\langle(\alpha, r) : ps, ds : a_1 : \ldots : a_n, es\rangle \vdash_{\mathsf{red}} \langle ps, ds : f_{\mathfrak{A}}(a_1, \ldots, a_n), es\rangle$ and $\mathsf{redsym}(\alpha) = [f]$.
  Again, ps must be connected because $(\alpha, r) : ps$ is.
- $\langle(\alpha, r) : ps, ds : \mathsf{true}, es\rangle \vdash_{\mathsf{red}} \langle(\alpha.2, d) : ps, ds, es\rangle$ and $\mathsf{redsym}(\alpha) = cond[t_1, t_2]$.
  Now, the assertion follows because $\mathsf{follow}(\alpha.2) = \mathsf{follow}(\alpha)$.
- $\langle(\alpha, r) : ps, ds : \mathsf{false}, es\rangle \vdash_{\mathsf{red}} \langle(\alpha.3, d) : ps, ds, es\rangle$ and $\mathsf{redsym}(\alpha) = cond[t_1, t_2]$.
  Analogously.
- $\langle(\alpha, r) : ps, ds : a_1 : \ldots : a_{\sigma_j}, e : es\rangle \vdash_{\mathsf{red}}$
  $\langle(j, d) : (r + 1, r) : ps, ds, (a_1, \ldots, a_{\sigma_j}, \alpha.(\sigma_j + 1), \ldots, \alpha.n_j) \cdot e : e : es\rangle$ and $\mathsf{redsym}(\alpha) = F_j[t_{\sigma_j+1}, \ldots, t_{n_j}]$.
  The assertion follows from $\mathsf{follow}(j) = r + 1$.
- $\langle(r + 1, r) : ps, ds, e : es\rangle \vdash_{\mathsf{red}} \langle ps, ds, es\rangle$.
  Necessarily, ps is connected. □

This address connection of program stack values of reachable address states shows that the program stack can be replaced by a program counter pointing to the next reduction symbol and a return stack which holds the dynamic first–address after a return address. The control of all other reduction symbols can be described statically by first and follow functions. These considerations suggest the following abstract machine for compiling recursive function definitions.

Definition 69 (Abstract stack machine): *Let $\mathfrak{A}$ be a branching Σ–algebra. Using the infinite set* $\mathbf{Adr} := \mathbb{N}^*$ *of addresses we define the abstract stack machine of $\mathfrak{A}$ by the set* $\mathbf{St}$ *of* ***states*** *together with the set* $\mathbf{Cmd}$ *of* ***commands*** *where*

$$\mathbf{St} := \mathbf{PC} \times \mathbf{RS} \times \mathbf{DS} \times \mathbf{ES}^{@}$$

and

$$\begin{aligned} \mathbf{PC} &:= \mathbf{Adr}, \\ \mathbf{RS} &:= \mathbf{Adr}^*, \\ \mathbf{DS} &:= A^*, \textit{ and} \\ \mathbf{ES}^{@} &:= (((A \cup \mathbf{Adr})^*)^*)^*, \end{aligned}$$

denote the sets of ***program counter****,* ***return stack****,* ***data stack****, and* ***environment stack*** *values, respectively, and where*

$$\begin{aligned} \mathbf{Cmd} := {} & \{\mathtt{EVAL}(x, \beta) \mid x \in \mathbb{X}, \beta \in \mathbf{Adr}\} \cup \\ & \{\mathtt{EXEC}(f, \beta) \mid f \in F_{\mathsf{base}}, \beta \in \mathbf{Adr}\} \cup \\ & \{\mathtt{SELECT}(\alpha_1, \alpha_2) \mid \alpha_1, \alpha_2 \in \mathbf{Adr}\} \cup \\ & \{\mathtt{CALL}(\alpha_0, \sigma, \alpha_1, \dots, \alpha_m, \beta) \mid \sigma, m \in \mathbb{N}, \alpha_i, \beta \in \mathbf{Adr}\} \cup \\ & \{\mathtt{RET}\}. \end{aligned}$$

Each command $\gamma \in \mathbf{Cmd}$ *denotes a* ***state transformation*** $[\![\gamma]\!] : \mathbf{St} \dashrightarrow \mathbf{St}$ *defined by*

- $[\![\mathtt{EVAL}(x_i, \beta)]\!]\langle pc, rs, ds, (z_1, \dots, z_k) \cdot e : es\rangle$
  $$:= \begin{cases} \langle \beta, rs, ds : z_i, (z_1, \dots, z_k) \cdot e : es\rangle & \textit{if } 1 \le i \le k \textit{ and } z_i \in A \\ \langle z_i, \beta : rs, ds, e : (z_1, \dots, z_k) \cdot e : es\rangle & \textit{if } 1 \le i \le k \textit{ and } z_i \in \mathbf{Adr} \end{cases}$$
- $[\![\mathtt{EXEC}(f, \beta)]\!]\langle pc, rs, ds : a_1 : \dots : a_n, es\rangle$
  $:= \langle \beta, rs, ds : f_{\mathfrak{A}}(a_1, \dots, a_n), es\rangle$
- $[\![\mathtt{SELECT}(\alpha_1, \alpha_2)]\!]\langle pc, rs, ds : b, es\rangle$
  $$:= \begin{cases} \langle \alpha_1, rs, ds, es\rangle & \textit{if } b = \mathsf{true} \\ \langle \alpha_2, rs, ds, es\rangle & \textit{if } b = \mathsf{false} \end{cases}$$
- $[\![\mathtt{CALL}(\alpha_0, \sigma, \alpha_1, \dots, \alpha_m, \beta)]\!]\langle pc, rs, ds : a_1 : \dots : a_\sigma, e : es\rangle$
  $:= \langle \alpha_0, \beta : rs, ds, (a_1, \dots, a_\sigma, \alpha_1, \dots, \alpha_m) \cdot e : e : es\rangle$
- $[\![\mathtt{RET}]\!]\langle pc, \alpha : rs, ds, e : es\rangle$
  $:= \langle \alpha, rs, ds, es\rangle.$

Note that this abstract stack machine operates only on A–values and addresses whereas terms have been eliminated completely.

Definition 70 (Machine programs): *Let* $\mathbf{Adr}_f \subseteq \mathbf{Adr}$ *be a non–empty and finite subset of addresses. A mapping*

$$\pi : \mathbf{Adr}_f \to \mathbf{Cmd}$$

is called a ***machine program****. For its semantics we associate with π a transition relation*

$$\vdash_\pi \subseteq \mathbf{St} \times \mathbf{St}$$

defined by $\quad st = \langle pc, rs, ds, es\rangle \vdash_\pi st' \quad$ *if* $\quad pc \in \mathbf{Adr}_f$ *and* $st' = [\![\pi(pc)]\!](st)$*.*

The task of translating a recursive function definition into a suitable machine program is now easily accomplished. We choose $\mathbf{Adr}_R^{\mathsf{ret}}$ as a finite address set. The corresponding commands are determined by their associated reduction symbols together with first and follow functions.

Definition 71 (Compiling a recursive function definition): *For* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ *we define its machine program*

$$\pi_R : \mathbf{Adr}_R^{\mathsf{ret}} \to \mathbf{Cmd}$$

by

$$\pi_R(\alpha) := \begin{cases} \mathtt{EVAL}(x_i, \mathsf{follow}(\alpha)) & \textit{if } \mathsf{redsym}(\alpha) = [x_i], \\ \mathtt{EXEC}(f, \mathsf{follow}(\alpha)) & \textit{if } \mathsf{redsym}(\alpha) = [f], \\ \mathtt{SELECT}(\mathsf{first}(\alpha.2), \mathsf{first}(\alpha.3)) & \textit{if } \mathsf{redsym}(\alpha) = \mathsf{cond}[t_1, t_2], \\ \mathtt{CALL}(\mathsf{first}(j), \sigma_j, \mathsf{first}(\alpha.(\sigma_j + 1)), & \\ \ldots, \mathsf{first}(\alpha.n_j), \mathsf{follow}(\alpha)) & \textit{if } \mathsf{redsym}(\alpha) = F_j[t_{\sigma_j+1}, \ldots, t_{n_j}], \\ \mathtt{RET} & \textit{if } \mathsf{redsym}(\alpha) = \mathsf{ret}. \end{cases}$$

We see that compiling essentially appears to be a finite abstraction of control: while the interpreter controls reduction steps by term decomposition during computation, the compiler generates an explicit control through first and follow functions independent of a particular computation. Only the `RET`–command requires a dynamic control through the return stack.

Iterating the transitions of machine programs we define a compiler semantics for recursive function definitions as follows.

Definition 72 (Compiler semantics): *For* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma^{(w,s)}$ *the* ***compiler semantics***

$$\mathsf{Cmp}[\![R]\!]_{\mathfrak{A}} : A^w \to A_\perp^s$$

is defined by:

$$\mathsf{Cmp}[\![R]\!]_{\mathfrak{A}}(a_1, \ldots, a_n) := \begin{cases} b & \textit{if } \langle \mathsf{first}(0), 0.0, \varepsilon, (a_1, \ldots, a_n) \rangle \vdash_{\pi_R}^* \langle 0.0, \varepsilon, b, \varepsilon \rangle \\ \perp^s & \textit{if no such } b \in A^s \textit{ exists} \end{cases}$$

Note that 0 marks the position of the initial term whereas $0.0 \notin \mathbf{Adr}_R$ is taken as a standard stop address.

Example 73: For our multiplication example $(R_{mult}, \mathfrak{N})$, given by

$$\begin{aligned} F &= \lambda^0\, x.G\,(x-1)\,(Hx) \\ G &= \lambda^1\, x\, y.\mathsf{cond}\,(x=0)\,x\,((G\,(x-1)\,y) + y) \\ H &= \lambda^0\, x.H\,(x+1), \end{aligned}$$

we get the following machine program $\pi_{R_{mult}}$ where we simply write $\alpha : \gamma;$ instead of $\pi_{R_{mult}}(\alpha) = \gamma$:

$$
\begin{aligned}
0 &: \mathtt{CALL}(1.1.1, 0, 0.1, 4);\\
0.1 &: \mathtt{EVAL}(x, 4);\\
1 &: \mathtt{CALL}(2.1.1, 1, 1.2, 4);\\
1.1 &: \mathtt{EXEC}(-, 1);\\
1.1.1 &: \mathtt{EVAL}(x, 1.1.2);\\
1.1.2 &: \mathtt{EXEC}(1, 1.1);\\
1.2 &: \mathtt{CALL}(3, 0, 1.2.1, 4);\\
1.2.1 &: \mathtt{EVAL}(x, 4);\\
2 &: \mathtt{SELECT}(2.2, 2.3.1.1.1);\\
2.1 &: \mathtt{EXEC}(=, 2);\\
2.1.1 &: \mathtt{EVAL}(x, 2.1.2);\\
2.1.2 &: \mathtt{EXEC}(0, 2.1);\\
2.2 &: \mathtt{EVAL}(x, 4);\\
2.3 &: \mathtt{EXEC}(+, 4);\\
2.3.1 &: \mathtt{CALL}(2.1.1, 1, 2.3.1.2, 2.3.2);\\
2.3.1.1 &: \mathtt{EXEC}(-, 2.3.1);\\
2.3.1.1.1 &: \mathtt{EVAL}(x, 2.3.1.1.2);\\
2.3.1.1.2 &: \mathtt{EXEC}(1, 2.3.1.1);\\
2.3.1.2 &: \mathtt{EVAL}(y, 4);\\
2.3.2 &: \mathtt{EVAL}(y, 2.3);\\
3 &: \mathtt{CALL}(3, 0, 3.1.1, 4);\\
3.1 &: \mathtt{EXEC}(+, 4);\\
3.1.1 &: \mathtt{EVAL}(x, 3.1.2);\\
3.1.2 &: \mathtt{EXEC}(1, 3.1);\\
4 &: \mathtt{RET};
\end{aligned}
$$

This machine program generates for input $x = \mathbf{1}$ the following stack machine computation where $\mathbf{0}$ and $\mathbf{1}$ are integers in contrast to the addresses 0 and 1:

$$
\begin{array}{llllllr}
\langle 0 & , & 0.0 & , \varepsilon & , & & (\mathbf{1})\,\rangle\\
\langle 1.1.1 & , & 4{:}0.0 & , \varepsilon & , & & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 0.1 & , & 1.1.2{:}4{:}0.0 & , \varepsilon & , & & (\mathbf{1}){:}(0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 4 & , & 1.1.2{:}4{:}0.0 & , \mathbf{1} & , & & (\mathbf{1}){:}(0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 1.1.2 & , & 4{:}0.0 & , \mathbf{1} & , & & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 1.1 & , & 4{:}0.0 & , \mathbf{1}{:}\mathbf{1} & , & & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 1 & , & 4{:}0.0 & , \mathbf{0} & , & & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 2.1.1 & , & 4{:}4{:}0.0 & , \varepsilon & , & (\mathbf{0}, 1.2)\cdot(0.1)\cdot(\mathbf{1}){:} & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 2.1.2 & , & 4{:}4{:}0.0 & , \mathbf{0} & , & (\mathbf{0}, 1.2)\cdot(0.1)\cdot(\mathbf{1}){:} & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 2.1 & , & 4{:}4{:}0.0 & , \mathbf{0}{:}\mathbf{0} & , & (\mathbf{0}, 1.2)\cdot(0.1)\cdot(\mathbf{1}){:} & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 2 & , & 4{:}4{:}0.0 & , \mathsf{true} & , & (\mathbf{0}, 1.2)\cdot(0.1)\cdot(\mathbf{1}){:} & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 2.2 & , & 4{:}4{:}0.0 & , \varepsilon & , & (\mathbf{0}, 1.2)\cdot(0.1)\cdot(\mathbf{1}){:} & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 4 & , & 4{:}4{:}0.0 & , \mathbf{0} & , & (\mathbf{0}, 1.2)\cdot(0.1)\cdot(\mathbf{1}){:} & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 4 & , & 4{:}0.0 & , \mathbf{0} & , & & (0.1)\cdot(\mathbf{1}){:}(\mathbf{1})\,\rangle\\
\langle 4 & , & 0.0 & , \mathbf{0} & , & & (\mathbf{1})\,\rangle\\
\langle 0.0 & , & \varepsilon & , \mathbf{0} & , & & \varepsilon\,\rangle
\end{array}
$$

This shows that $\mathsf{Cmp}[\![R_{mult}]\!]\mathfrak{N}(1) = 0$.

Now we are well prepared for the final step of our compiler correctness proof.

Theorem 74 (Compiler correctness): *For every* $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$, *compiler and address interpreter semantics coincide:*

$$\mathsf{Cmp}[\![R]\!]_{\mathfrak{A}} = \mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}}.$$

Proof: In order to correlate address interpreter computations with abstract machine computations we abstract from an address state an abstract machine state:

$$\mathsf{mst} : \mathbf{IntSt}^{@} \to \mathbf{St}$$

is defined by

$$\begin{aligned}
\mathsf{mst}\langle ps, ds, es\rangle &:= \langle \mathsf{pc}(ps), \mathsf{rs}(ps), ds, \mathsf{es}(es)\rangle \\
\mathsf{pc}(\varepsilon) &:= 0.0 \\
\mathsf{pc}((\alpha, d) : ps) &:= \mathsf{first}(\alpha) \\
\mathsf{pc}((\alpha, r) : ps) &:= \alpha \\
\mathsf{rs}(\varepsilon) &:= \varepsilon \\
\mathsf{rs}((\alpha, _) : ps) &:= \mathsf{rs}(ps) \text{ if } \alpha \neq r + 1 \\
\mathsf{rs}((r + 1, r) : ps) &:= \mathsf{pc}(ps) : \mathsf{rs}(ps) \\
\mathsf{es}(e_1 : \ldots : e_n) &:= \mathsf{es}(e_1) : \ldots : \mathsf{es}(e_n) \\
\mathsf{es}(\bar{z_1} \cdot \ldots \cdot \bar{z_m}) &:= \mathsf{es}(\bar{z_1}) \cdot \ldots \cdot \mathsf{es}(\bar{z_m}) \\
\mathsf{es}(z_1, \ldots, z_k) &:= (\mathsf{es}(z_1), \ldots, \mathsf{es}(z_k)) \\
\mathsf{es}(z) &:= \begin{cases} z & \text{if } z \in A \\ \mathsf{first}(z) & \text{if } z \in \mathbf{Adr}_R. \end{cases}
\end{aligned}$$

We prove by induction on reachable address states that mst maps an address interpreter computation onto a semantically equivalent abstract machine computation. Here we exploit the fact that reachable address states have connected program stack values.

(i) The initial address state for $(a_1, \ldots, a_n)$ is mapped onto the corresponding initial machine state:
$\mathsf{mst}\langle (0, d) : (r + 1, r), \varepsilon, (a_1, \ldots, a_n)\rangle = \langle \mathsf{first}(0), 0.0, \varepsilon, (a_1, \ldots, a_n)\rangle$.

(ii) A decomposition step between address states does not change the corresponding machine states. There are two cases:
   (a) If $st = \langle (\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle (\alpha.1, d) : ps_1), ds, es\rangle = st'$, it follows that $\mathsf{first}(\alpha) = \mathsf{first}(\alpha.1)$ and $\mathsf{rs}((\alpha, d) : ps) = \mathsf{rs}((\alpha.1, d) : ps_1)$ and therefore that $\mathsf{mst}(st) = \mathsf{mst}(st')$.
   (b) If $st = \langle (\alpha, d) : ps, ds, es\rangle \vdash_{\mathsf{dec}} \langle (\alpha, r) : ps), ds, es\rangle = st'$, then $\mathsf{first}(\alpha) = \alpha$ and $\mathsf{rs}((\alpha, d) : ps) = \mathsf{rs}((\alpha, r) : ps))$, so that again $\mathsf{mst}(st) = \mathsf{mst}(st')$.

(iii) Reduction steps between reachable address states are mapped onto machine transitions. As for a reachable state $st = \langle (\alpha, r) : ps, ds, es\rangle$ the program stack value $(\alpha, r) : ps$ is connected, it follows that $ps \neq \varepsilon$ and $\mathsf{follow}(\alpha) = \mathsf{pc}(ps)$ provided that $\alpha \neq r + 1$.
Now, if $st \vdash_{\mathsf{red}} st'$ and st is reachable, then $\mathsf{mst}(st) \vdash_{\pi_R} \mathsf{mst}(st')$. This is proved by the following case analysis.

(a) Let $st = \langle(\alpha, r) : ps, ds, (z_1, \ldots, z_k) \cdot e : es\rangle$, $\mathsf{redsym}(\alpha) = [x_i]$ and $z_i \in A$. It follows that $st \vdash_{\mathsf{red}} st' = \langle ps, ds : z_i, (z_1, \ldots, z_k) \cdot e : es\rangle$. Since $\pi_R(\alpha) = \mathtt{EVAL}(x_i, \mathsf{follow}(\alpha))$, we get for the corresponding machine states

$$\begin{aligned}
\mathsf{mst}(st) &= \langle\alpha, \mathsf{rs}(ps), ds, \mathsf{es}((z_1, \ldots, z_k) \cdot e : es)\rangle \\
&\vdash_{\pi_R} [\![\mathtt{EVAL}(x_i, \mathsf{follow}(\alpha))]\!](\mathsf{mst}(st)) \\
&= \langle\mathsf{follow}(\alpha), \mathsf{rs}(ps), ds : z_i, \mathsf{es}((z_1, \ldots, z_k) \cdot e : es)\rangle \\
&= \langle\mathsf{pc}(ps), \mathsf{rs}(ps), ds : z_i, \mathsf{es}((z_1, \ldots, z_k) \cdot e : es)\rangle \\
&= \mathsf{mst}(st').
\end{aligned}$$

(b) If $st = \langle(\alpha, r) : ps, ds, (z_1, \ldots, z_k) \cdot e : es\rangle$, $\mathsf{redsym}(\alpha) = [x_i]$ and $z_i \in \mathbf{Adr}_R$, it follows that $st \vdash_{\mathsf{red}} st' = \langle(z_i, d) : (r + 1, r) : ps, ds, e : (z_1, \ldots, z_k) \cdot e : es\rangle$ and we conclude similarly

$$\begin{aligned}
\mathsf{mst}(st) &\vdash_{\pi_R} \langle\mathsf{first}(z_i), \mathsf{follow}(\alpha) : \mathsf{rs}(ps), ds, \mathsf{es}(e : (z_1, \ldots, z_k) \cdot e : es)\rangle \\
&= \mathsf{mst}(st').
\end{aligned}$$

(c) For $st = \langle(\alpha, r) : ps, ds : a_1 : \ldots : a_n, es\rangle$ with $\mathsf{redsym}(\alpha) = [f]$ it follows that $st \vdash_{\mathsf{red}} st' = \langle ps, ds : f_{\mathfrak{A}}(a_1, \ldots, a_n), es\rangle$ and $\pi_R(\alpha) = \mathtt{EXEC}(f, \mathsf{follow}(\alpha))$, so that

$$\begin{aligned}
\mathsf{mst}(st) &= \langle\alpha, \mathsf{rs}(ps), ds : a_1 : \ldots : a_n, \mathsf{es}(es)\rangle \\
&\vdash_{\pi_R} [\![\mathtt{EXEC}(f, \mathsf{follow}(\alpha))]\!](\mathsf{mst}(st)) \\
&= \langle\mathsf{follow}(\alpha), \mathsf{rs}(ps), ds : f_{\mathfrak{A}}(a_1, \ldots, a_n), \mathsf{es}(es)\rangle \\
&= \langle\mathsf{pc}(ps), \mathsf{rs}(ps), ds : f_{\mathfrak{A}}(a_1, \ldots, a_n), \mathsf{es}(es)\rangle \\
&= \mathsf{mst}(st').
\end{aligned}$$

(d) If $st = \langle(\alpha, r) : ps, ds : \mathsf{true}, es\rangle$ and $\mathsf{redsym}(\alpha) = \mathsf{cond}[t_1, t_2]$, we see that $st \vdash_{\mathsf{red}} st' = \langle(\alpha.2, d) : ps, ds, es\rangle$ and $\pi_R(\alpha) = \mathtt{SELECT}(\mathsf{first}(\alpha.2), \mathsf{first}(\alpha.3))$, so that

$$\begin{aligned}
\mathsf{mst}(st) &= \langle\alpha, \mathsf{rs}(ps), ds : \mathsf{true}, \mathsf{es}(es)\rangle \\
&\vdash_{\pi_R} [\![\mathtt{SELECT}(\mathsf{first}(\alpha.2), \mathsf{first}(\alpha.3))]\!](\mathsf{mst}(st)) \\
&= \langle\mathsf{first}(\alpha.2), \mathsf{rs}(ps), ds, \mathsf{es}(es)\rangle \\
&= \mathsf{mst}(st').
\end{aligned}$$

In the false–case the same holds with the second alternative.

(e) If $st = \langle(\alpha, r) : ps, ds : a_1 : \ldots : a_{\sigma_j}, e : es\rangle$ and $\mathsf{redsym}(\alpha) = F_j[t_{\sigma_j+1}, \ldots, t_{n_j}]$, we have $st \vdash_{\mathsf{red}}$ $st' = \langle(j, d) : (r + 1, r) : ps, ds, (a_1, \ldots, a_{\sigma_j}, \alpha.(\sigma_j + 1), \ldots, \alpha.n_j) \cdot e : e : es\rangle$. As $\pi_R(\alpha) = \mathtt{CALL}(\mathsf{first}(j), \sigma_j, \mathsf{first}(\alpha.(\sigma_j + 1)), \ldots, \mathsf{first}(\alpha.n_j), \mathsf{follow}(\alpha))$, it follows that

$$\begin{aligned}
\mathsf{mst}(st) &= \langle\alpha, \mathsf{rs}(ps), ds : a_1 : \ldots : a_{\sigma_j}, \mathsf{es}(e : es)\rangle \\
&\vdash_{\pi_R} [\![\mathtt{CALL}(\mathsf{first}(j), \sigma_j, \mathsf{first}(\alpha.(\sigma_j + 1)), \ldots, \mathsf{follow}(\alpha))]\!](\mathsf{mst}(st)) \\
&= \langle\mathsf{first}(j), \mathsf{follow}(\alpha) : \mathsf{rs}(ps), ds, \\
&\qquad \mathsf{es}((a_1, \ldots, a_{\sigma_j}, \alpha.(\sigma_j + 1), \ldots, \alpha.n_j) \cdot e : e : es)\rangle \\
&= \mathsf{mst}(st').
\end{aligned}$$

(f) Finally, if $st = \langle (r+1,r) : ps, ds, e : es\rangle$, it holds that $st \vdash_{\mathsf{red}} st' = \langle ps, ds, es\rangle$. Since $\pi_R(r+1) = \mathtt{RET}$, the assertion follows by

$$\begin{aligned}\mathsf{mst}(st) &= \langle r+1, \mathsf{pc}(ps) : \mathsf{rs}(ps), ds, \mathsf{es}(e : es)\rangle \\ &\vdash_{\pi_R} [\![\mathtt{RET}]\!](\mathsf{mst}(st)) \\ &= \langle \mathsf{pc}(ps), \mathsf{rs}(ps), ds, \mathsf{es}(es)\rangle \\ &= \mathsf{mst}(st').\end{aligned}$$

These results imply that in fact address interpreter and compiler semantics coincide: if $\mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}}(a_1,\ldots,a_n) = b \in A$, then there is an address interpreter computation $\langle (0,d) : (r+1,r), \varepsilon, (a_1,\ldots,a_n)\rangle \vdash^* \langle \varepsilon, b, \varepsilon\rangle$. The corresponding abstract machine computation $\langle \mathsf{first}(0), 0.0, \varepsilon, (a_1,\ldots,a_n)\rangle \vdash^*_{\pi_R} \langle 0.0, \varepsilon, b, \varepsilon\rangle$ proves that also $\mathsf{Cmp}[\![R]\!]_{\mathfrak{A}}(a_1,\ldots,a_n) = b$ holds.
If on the other hand $\mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}}(a_1,\ldots,a_n) = \perp$, there is an infinite address interpreter computation $(st_i \vdash st_{i+1} \mid i \in \mathbb{N})$ starting from $st_0 = \langle (0,d) : (r+1,r), \varepsilon, (a_1,\ldots,a_n)\rangle$. It must contain an infinite number of reduction steps so that the corresponding machine computation starting from $\langle \mathsf{first}(0), 0.0, \varepsilon, (a_1,\ldots,a_n)\rangle$ is infinite, too. Hence, $\mathsf{Cmp}[\![R]\!]_{\mathfrak{A}}(a_1,\ldots,a_n) = \perp$. □

6 Conclusion

Using an algebraic and order–theoretic framework we presented a complete correctness proof for compiling recursive function definitions with strictness information into stack code. Due to the absence of higher–order functions and data structures we could develop a stack technique that avoids heaps and closures even for implementing lazy evaluation. Starting from a denotational view we defined a fixed–point semantics taking strictness information into account. For an operational view we first gave a non–deterministic single–step reduction semantics which could later be specialized to a deterministic left–reduction semantics integrating call–by–name and call–by–value evaluation. This separation of reduction semantics turned out to be essential for the equivalence proof being more complex than in the big–step case. Left–reductions naturally led to an interpreter which could thereafter be transformed appropriately into a compiler.

Altogether we proved the equivalence of all semantic models. For any recursive function definition $(R, \mathfrak{A}) \in \mathbf{Rfd}_\Sigma$ it holds that

$$\mathsf{Fp}[\![R]\!]_{\mathfrak{A}} = \mathsf{Red}[\![R]\!]_{\mathfrak{A}} = \mathsf{IRd}[\![R]\!]_{\mathfrak{A}} = \mathsf{Int}[\![R]\!]_{\mathfrak{A}} = \mathsf{Int}^{@}[\![R]\!]_{\mathfrak{A}} = \mathsf{Cmp}[\![R]\!]_{\mathfrak{A}}.$$

7 Historical remarks

Already S.C. Kleene [Kle52] presented in his first recursion theorem a connection between operational and fixed–point semantics. He considered strict arithmetical functions and used a computation that corresponds to call–by–value reduction. With the development of programming languages the practical importance of recursive function definitions became evident. As an early example we mention LISP. Its theoretical foundations were developed by J. McCarthy in [McC60]. The denotational method defining the meaning of a program by induction on its syntactic structure was introduced by D. Scott and C. Strachey [SS72]. D. Scott realized the importance of topological tools, in particular continuous functions on complete partial orders [Sco70]. The algebraic character of denotational semantics has been worked out most explicitly by J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright [GTWW77]. The relationship between fixed–point semantics and evaluation strategies was studied by J. Cadiou [Cad72], Z. Manna and J. Vuillemin [MV72]. However, they only considered the call–by–name fixed point as valid and viewed the call–by–value computation as incorrect when compared to fixed–point semantics. It was shown by J. de Bakker [Bak76] that there is also a fixed–point semantics corresponding to call–by–value. In his book [Win93] G. Winskel gives equivalence proofs for both cases, but using big–step instead of small–step semantics. Here, we consider the latter as it is more convenient for proving the correctness of stack code. The use of stacks for the implementation of programming languages dates back to the early work of F.L. Bauer and K. Samelson [SB60]. They suggested a stack for evaluating arithmetical expressions. The corresponding stack code was proved correct by J. McCarthy and J. Painter [MP67]. Although stacks turned out to be of central importance for compilers, the additional use of a heap was unavoidable. In this paper we could demonstrate that at least in the first order case heaps are unnecessary provided that we can read the full stack. In his master's thesis [Sch87] R. Schlör gave separate proofs for the correctness of call–by–name and call–by–value stack code using a closure technique.

References

[Bak76] J.W. de Bakker. Least fixed points revisited. *Theoretical Computer Science*, 2:155–181, 1976.

[Cad72] J. Cadiou. *Recursive Definitions of Partial Functions and Their Computation*. PhD thesis, Stanford University, 1972.

[Chi97] O. Chitil. The ς–semantics: A comprehensive semantics for functional programs. *Fundamenta Informaticae*, 31:253–294, 1997.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. North–Holland, Amsterdam, 1952.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3:184–195, 1960.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. *Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science*, 19:33–41, 1967.

[MV72] Z. Manna and J. Vuillemin. Fixpoint approach to the theory of computation. *Communications of the ACM*, 15(7):528–536, July 1972.

[SB60] K. Samelson and F.L. Bauer. Sequential formula translation. *Comm. ACM*, 3(2):76–83, 1960.

[Sch87] R. Schlör. Korrektheit der Übersetzung rekursiver Funktionsdefinitionen in Stackcode. Master's thesis, RWTH Aachen University, 1987.

[Sco70] D. Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, volume 3, pages 169–176, 1970.

[SS72] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proc. Symp. Computers and Automata*, pages 19–46. Wiley, New York, 1972.

[Tar55] A. Tarski. A lattice–theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[Wad96] P. Wadler. Lazy versus strict. *ACM Computing Surveys*, 28(2):318–320, June 1996.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

1987-01 * Fachgruppe Informatik: Jahresbericht 1986
1987-02 * David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 * Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 * Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 * Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 * Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 * Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 * Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 * Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 * Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 * Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12 J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 * Gabriele Esser, Johannes Rückert, Frank Wagner: Gesellschaftliche Aspekte der Informatik
1988-02 * Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 * Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 * Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 * Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 * Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 * Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 * Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 * W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 * Kai Jakobs: Towards User-Friendly Networking
1988-11 * Kai Jakobs: The Directory - Evolution of a Standard
1988-12 * Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 * Martine Schümmer: RS-511, a Protocol for the Plant Floor

1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing

1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation

1988-16 * Fachgruppe Informatik: Jahresbericht 1987

1988-17 * Wolfgang Thomas: Automata on Infinite Objects

1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs

1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers

1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen

1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment

1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers

1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies

1988-24 * Uschi Heuter: Generalized Definite Tree Languages

1989-01 * Fachgruppe Informatik: Jahresbericht 1988

1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik

1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions

1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)

1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?

1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems

1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control

1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial

1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation

1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science

1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?

1989-13 * Martine Schümmer: CNC/DNC Communication with MAP

1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30 T. Margaria: First-Order theories for the verification of complex FSMs
1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991
1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes

1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems

1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line

1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme

1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management

1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines

1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking

1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD

1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars

1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)

1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik

1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual

1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems

1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages

1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)

1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine

1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus

1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions

1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code

1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine

1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)

1992-19-08 D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus

1992-19-09 D. Howe, G. Burn: Experiments with strict STG code

1992-19-10 J. Glauert: Parallel Implementation of Functional Languages Using Small Processes

1992-19-11 M. Joy, T. Axford: A Parallel Graph Reduction Machine

1992-19-12 A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)

1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer

1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine

1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell

1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation

1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages

1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)

1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment

1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)

1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers

1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)

1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)

1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting

1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)

1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models

1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures

1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)

1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language

1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language

1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing

1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free

1992-24 K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications

1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification

1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety

1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design

1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zuendorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels
1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages
1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager
1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept
1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain
1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes
1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing
1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel
1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control
1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle
1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993
1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications
1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse
1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations
1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics
1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations
1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository
1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments
1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems
1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems
1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars
1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems
1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition
1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents
1994-16 P. Klein: Designing Software with Modula-3
1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems

1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming

1996-01 * Jahresbericht 1995

1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees

1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability

1996-05 Klaus Pohl: Requirements Engineering: An Overview

1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools

1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs

1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics

1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming

1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents

1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines

1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation

1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell

1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems

1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming

1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management

1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement

1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models

1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools

1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases

1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation

1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations

1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

1998-01 * Fachgruppe Informatik: Jahresbericht 1997

1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments

1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future
1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
1998-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen
1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases
1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML
1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
1999-01 * Jahresbericht 1998
1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager
1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference
1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie
1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL
1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge, Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 Mareike Schoop: Cooperative Document Management
2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 * Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08 Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 * Jahresbericht 2000
2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs

2004-01 * Fachgruppe Informatik: Jahresbericht 2004

2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

* These reports are only available as a printed version.

Please contact biblio@informatik.rwth-aachen.de to obtain copies.