INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

Eine semantische Analyse struktureller Rekursion

Bearbeiter:	Andreas Abel
Aufgabensteller:	Prof. Dr. Peter Clote, Ph.D.
Betreuer:	Dr. Thorsten Altenkirch, Ph.D.

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Diplomarbeit

Eine semantische Analyse struktureller Rekursion

Bearbeiter: Andreas Abel
Aufgabensteller: Prof. Dr. Peter Clote, Ph.D.
Betreuer: Dr. Thorsten Altenkirch, Ph.D.
Abgabetermin: 26. Februar 1999

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 26. Februar 1999

....

(Unterschrift des Kandidaten)

Danksagung

An erster Stelle danke ich Jesus Christus, der das Universum und den menschlichen Geist geformt hat und damit auch die Mathematik und Technik, die wir schaffen und zu unserem Wohl gebrauchen können. Auch hat er an einer kritischen Stelle im Laufe dieser Arbeit mein Gebet gehört und mir geholfen.

Sodann danke ich Thorsten Altenkirch, der in unkomplizierter Weise meine Arbeit betreut und sich viel Zeit für meine Fragen genommen hat. Ebenso Ralph Matthes, der die Arbeit gründlich gelesen hat und mich an seinem typentheoretischen Erfahrungsschatz teilhaben ließ, sofern es sein mit Doktorarbeit und Rigorosum gefüllter Terminkalender erlaubte. Und Rolf Backofen, der mir in einigen technischen Fragen geholfen hat.

Meinen Eltern danke ich für die wohlwollende Finanzierung meines Studiums und dafür, dass sie immer für mich da sind.

Und schließlich danke ich Julia, dass sie in meinem Leben aufgetaucht ist (obwohl sie mich eigentlich von meiner Arbeit nur abgelenkt hat).

A Semantic Analysis of Structural Recursion

Abstract

We consider a type theoretic language with lambda abstraction, disjoint unions, pairs and recursive terms and show that all structurally recursive functions terminate, i.e. functions, that contain recursive calls only with a structurally smaller argument. To this end, we define a semantics $[\![\sigma]\!]$ over all types and show that it is wellfounded w.r.t. the structural ordering on the values (normal forms of our terms). Then by wellfounded induction we can easily infer termination of a recursive function at all inputs in $[\![\sigma]\!]$ from termination at all smaller inputs. Later we extend our solution from strictly positive types to positive types and mutual recursive functions.

Contents

1 Introduction

By the seventh day God had finished the work he had been doing; so on the seventh day he rested from all his work. And God blessed the seventh day and made it holy, because on it he rested from all the work of creating that he had done.
The Bible, NIV, Genesis 2:2,3

In the research field of program verification termination is an important aspect. In general the negative solution of Hilbert's halting problem showed that termination is undecidable. Nevertheless much work has been done on the question of termination, especially in the field of term rewriting systems [Der87, Ste95] – where several termination criteria have been developed – and more recently in the field of functional programming languages, e.g. [BG96, Gie97], to which our work contributes. We prove the following termination criterium:

> All structurally recursive functions terminate.

We define a *structurally recursive* function as

> a function f that terminates for all arguments v, if it does so under the condition that it terminates for all structurally smaller arguments v'.

Our termination criterium only (and then trivially) holds if the domain of f is wellfounded w.r.t. the structural ordering $<$, i.e. if there are no infinite descending chains $v > v' > \ldots$.

Functional programming languages like SML [Pau91] and Haskell [HPF97] and implementations of type theory like ALF [AGNvS94] and LEGO [LP92] allow definition of recursive functions/terms by pattern matching [Coq92]. A function is structurally recursive if

1. the pattern is total (i.e. no possible case is omitted) and
2. recursive calls use only structurally smaller arguments.

MuTTI (Munich Type Theory Implementation) is a functional language with dependent types. Pattern matching is done via case analysis of only one expression (no combined patterns), which rids us of a complicated totality check for condition 1.

For the simply typed sublanguage of MuTTI named foetus, we have developed a program which checks condition 2., where we obtain structurally smaller arguments only by case analysis, projection and function application. This restriction makes *being structurally smaller* a syntactic property that can easily be checked by foetus. For mutually recursive functions with multiple arguments, foetus constructs a call graph and tries to determine an permutation of the arguments with respect to which termination is verified using the lexicographic ordering (for detailed explanation see [Abe98]).

Our task now is to prove the soundness of our method, i.e. of our termination criterium. Therefore we consider a system with unit, sum, product, function and strictly positive recursive types that has the same expressivity as foetus, and we restrict to functions of one parameter without mutual recursion. After defining the types and terms of this system (section 2) we specify an operational semantics (section 3) and semantical values (section 4) on which we define a structural ordering. After having proven wellfoundedness of this ordering (section 5) we easily obtain the soundness of our termination criterium (section 6). We then extend our proof to systems with positive types (section 7.1), functions with multiple parameters (section 7.2) and mutual recursion (section 7.3) and we conclude by discussing further work (section 8).

A comment on our method for the reader familiar with normalization proofs. Our method of proving normalization for our system is a very intuitive one. Since we do not define a reduction relation as it is common for the lambda calculus in the literature, but treat it as a functional programming language with a fixed evaluation strategy, we neither have to consider terms of open types (i.e. types with free variables), nor show *strong* normalization, nor make use of sophisticated concepts like Girard's *candidates of reducibility* or saturated sets [Gir72, Alt93]. Our proof is straight and simple.

2 The language

We need a language that is strong enough to form typical functional programs like functions on natural numbers, lists and user defined data types. On the other hand it has to be simple enough to enable reasoning about it.

2.1 Types

The type system we have chosen is constructed from the base type 1 (representing the set with one element), type variables $X, Y, Z, \ldots$ and the type constructors $+$ (sum), $\times$ (product), $\rightarrow$ (function space) and Rec (recursive type). Within this system the set of natural numbers may be expressed as $N := \mathrm{Rec}\, X.1 + X$ and the set of lists over natural numbers as $ListN := \mathrm{Rec}\, X.1 + N \times X$. In the literature recursive types are often called μ-types.

To simplify the central proof of wellfoundedness (section 5) we restrict ourselves to strictly positive recursive types now. Later we will extend our system to positive recursive types (section 7.1).

Type variables. Assume a countably infinite set of type variables $\mathrm{TyVars} = \{X, Y, Z, \ldots\}$. A type variable X appears *strictly positive* within a type τ iff it appears never on the left side of some $\rightarrow$ (see rule (Arr) below).

Notation. For convenience we mix list, tuple and set notation: By $\vec{X}$ we mean a (possibly empty) list of type variables ($\vec{X} \subset \mathrm{TyVars}$) without duplicates (an ordered set) and by $\vec{X}, X$ the appending of X to $\vec{X}$ assuming $X \notin \vec{X}$.

Types. We inductively define the family of sets of types $\mathrm{Ty}(\vec{X})$ indexed over a finite set of type variables $\vec{X} \subset \mathrm{TyVars}$ appearing only strictly positive as follows:

$$\text{(Unit)} \quad \frac{}{1 \in \mathrm{Ty}(\emptyset)}$$

$$\text{(Var)} \quad \frac{\vec{X}, X \subset \mathrm{TyVars}}{X \in \mathrm{Ty}(\vec{X}, X)} \qquad \text{(Weak)}^1 \quad \frac{\sigma \in \mathrm{Ty}(\vec{X}) \quad X \notin \vec{X}}{\sigma \in \mathrm{Ty}(\vec{X}, X)}$$

$$\text{(Sum)} \quad \frac{\sigma, \tau \in \mathrm{Ty}(\vec{X})}{\sigma + \tau \in \mathrm{Ty}(\vec{X})} \qquad \text{(Prod)} \quad \frac{\sigma, \tau \in \mathrm{Ty}(\vec{X})}{\sigma \times \tau \in \mathrm{Ty}(\vec{X})}$$

$$\text{(Arr)} \quad \frac{\sigma \in \mathrm{Ty}(\emptyset) \quad \tau \in \mathrm{Ty}(\vec{X})}{\sigma \to \tau \in \mathrm{Ty}(\vec{X})} \qquad \text{(Rec)} \quad \frac{\sigma \in \mathrm{Ty}(\vec{X}, X)}{\mathrm{Rec}\, X.\sigma \in \mathrm{Ty}(\vec{X})}$$

We can restrict the free type variables to strictly positive ones because unlike Girards System F [Gir72], we have no polymorphic types and therefore need type variables only to construct recursive types.

Notation. In the following we write $\sigma(\vec{X})$ to express $\sigma \in \mathrm{Ty}(\vec{X})$. Then σ and $\sigma(\vec{X})$ are synonyms. We also abbreviate the set of closed types $\mathrm{Ty}(\emptyset)$ to Ty.

Renaming convention for types. Rec binds a type variable X in a type $\sigma(\vec{X}, X)$, and we may replace all appearances of X in $\mathrm{Rec}\, X.\sigma(\vec{X}, X)$ by any new variable $Y \notin \vec{X}$ without altering the actually denoted type. Thus we do not distinguish between $\mathrm{Rec}\, X.\sigma(\vec{X}, X)$ and $\mathrm{Rec}\, Y.\sigma(\vec{X}, Y)$.

Our style of variable introduction and binding (see rules (Var), (Weak) and (Rec) below) is very near to an implementation of variables by deBruijn-indices (see [dB72]).[2] This saves us a lot of work in defining substitution, but one must become familiar with the notation, e.g. that (Weak) is an explicit type (and later term) constructor. As compromise we have kept the variable names for better readability and thus formally need a renaming convention.

Substitution. Now we can define simultaneous substitution on types in the usual way. Provided a type $\sigma(\vec{X})$ with a list of free variables $\vec{X}$ and

[1] Unlike all other type forming rules (Weak) does not alter the term describing the type in our representation. We have left out a constructor like $\mathrm{Wk}X.\sigma$ for better readability. Our implementation in LEGO, however, uses this explicit constructor.

[2] That means that one enumerates the variables and obtains rules like (Var) $n \in \mathrm{Ty}(n+1)$, (Weak) $\mathrm{Ty}(n) \subset \mathrm{Ty}(n+1)$ and (Rec) $\sigma \in \mathrm{Ty}(n+1) \Rightarrow \mathrm{Rec}\, n{+}1.\, \sigma \in \mathrm{Ty}(n)$.

an equally long list of types $\vec{\rho} \in \mathrm{Ty}(\vec{Y})$ we define $\sigma[\vec{X} := \vec{\rho}] \in \mathrm{Ty}(\vec{Y})$ by recursion over σ as follows:

(Unit) $1[] := 1$

(Var) $X[\vec{X} := \vec{\rho}, X := \rho] := \rho$

(Weak)[3] $\sigma(\vec{X}, X)[\vec{X}, X := \vec{\rho}, \rho] := \sigma(\vec{X})[\vec{X} := \vec{\rho}]$

(Sum) $\left(\sigma(\vec{X}) + \tau(\vec{X})\right)[\vec{X} := \vec{\rho}] := \sigma[\vec{X} := \vec{\rho}] + \tau[\vec{X} := \vec{\rho}]$

(Prod) $\left(\sigma(\vec{X}) \times \tau(\vec{X})\right)[\vec{X} := \vec{\rho}] := \sigma[\vec{X} := \vec{\rho}] \times \tau[\vec{X} := \vec{\rho}]$

(Arr) $\left(\sigma(\emptyset) \to \tau(\vec{X})\right)[\vec{X} := \vec{\rho}] := \sigma \to (\tau[\vec{X} := \vec{\rho}])$

(Rec) We can assume $Z \notin \vec{Y}$ by the renaming convention, hence we can weaken all types $\rho_i(\vec{Y})$ by Z: $\left(\mathrm{Rec}\, Z.\sigma(\vec{X}, Z)\right)[\vec{X} := \vec{\rho}] := \mathrm{Rec}\, Z.\left(\sigma[\vec{X} := \vec{\rho}(\vec{Y}, Z), Z := Z]\right)$

Notation. We write $\sigma(\vec{\rho})$ for $\sigma(\vec{X})[\vec{X} := \vec{\rho}]$ and define substitution of a single variable Y as $\sigma(\vec{X}, Y, \vec{Z})[Y := \rho] := \sigma[\vec{X} := \vec{X}, Y := \rho, \vec{Z} := \vec{Z}]$, which we further abbreviate to $\sigma(\vec{X}, \rho, \vec{Z})$.

2.2 Terms

Now we define the terms inhabiting the above defined types. The definitions are very similar to a typed lambda calculus enriched by sums and products, except for the recursive terms we allow. We have decided to type terms over contexts to easily define closures afterwards.

Term variables and contexts. Assume a countably infinite set of term variables $\mathrm{TmVars} = \{g, x, y, z, \ldots\}$. Provided the closed types $\sigma_1, \ldots \sigma_n$ we can form a *context* $\Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \in \mathrm{Cxt}$ as a list of pairwise distinct term variables together with their types. We write x^σ to express the variable is of type σ in a given context.

[3] Note that in the weakened type σ the newly introduced variable X does not appear. Therefore the right side is well defined.

Terms. We define the set of terms $\mathrm{Tm}^\sigma[\Gamma]$ of a closed type σ over context Γ inductively as follows:

$$\text{(unit)}\quad \frac{}{() \in \mathrm{Tm}^1[]}$$

$$\text{(var)}\quad \frac{\Gamma \in \mathrm{Cxt} \quad x \notin \Gamma}{x \in \mathrm{Tm}^\sigma[\Gamma, x^\sigma]} \qquad \text{(weak)}\quad \frac{t \in \mathrm{Tm}^\sigma[\Gamma] \quad x \notin \Gamma}{t \in \mathrm{Tm}^\sigma[\Gamma, x^\tau]}$$

$$\text{(inl)}\quad \frac{t \in \mathrm{Tm}^\sigma[\Gamma] \quad \tau \in \mathrm{Ty}}{\mathrm{inl}(t) \in \mathrm{Tm}^{\sigma+\tau}[\Gamma]} \qquad \text{(inr)}\quad \frac{t \in \mathrm{Tm}^\tau[\Gamma] \quad \sigma \in \mathrm{Ty}}{\mathrm{inr}(t) \in \mathrm{Tm}^{\sigma+\tau}[\Gamma]}$$

$$\text{(case)}\quad \frac{t \in \mathrm{Tm}^{\sigma+\tau}[\Gamma] \quad l \in \mathrm{Tm}^\rho[\Gamma, x^\sigma] \quad r \in \mathrm{Tm}^\rho[\Gamma, y^\tau]}{\mathrm{case}(t, x^\sigma.l, y^\tau.r) \in \mathrm{Tm}^\rho[\Gamma]}$$

$$\text{(pair)}\quad \frac{s \in \mathrm{Tm}^\sigma[\Gamma] \quad t \in \mathrm{Tm}^\tau[\Gamma]}{(s, t) \in \mathrm{Tm}^{\sigma\times\tau}[\Gamma]}$$

$$\text{(fst)}\quad \frac{p \in \mathrm{Tm}^{\sigma\times\tau}[\Gamma]}{\mathrm{fst}(p) \in \mathrm{Tm}^\sigma[\Gamma]} \qquad \text{(snd)}\quad \frac{p \in \mathrm{Tm}^{\sigma\times\tau}[\Gamma]}{\mathrm{snd}(p) \in \mathrm{Tm}^\tau[\Gamma]}$$

$$\text{(lam)}\quad \frac{t \in \mathrm{Tm}^\tau[\Gamma, x^\sigma]}{\lambda x^\sigma.t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma]} \qquad \text{(rec)}\quad \frac{t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma, g^{\sigma\to\tau}]}{\mathrm{rec}\, g^{\sigma\to\tau}.t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma]}$$

$$\text{(app)}\quad \frac{t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma] \quad s \in \mathrm{Tm}^\sigma[\Gamma]}{t\, s \in \mathrm{Tm}^\tau[\Gamma]}$$

$$\text{(fold)}\quad \frac{t \in \mathrm{Tm}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}[\Gamma]}{\mathrm{fold}(t) \in \mathrm{Tm}^{\mathrm{Rec}\, X.\sigma(X)}[\Gamma]} \quad \text{(unfold)}\quad \frac{t \in \mathrm{Tm}^{\mathrm{Rec}\, X.\sigma(X)}[\Gamma]}{\mathrm{unfold}(t) \in \mathrm{Tm}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}[\Gamma]}$$

There are two main kinds of term forming rules: Rules for introducing types (the constructors (unit), (inl), (inr), (pair), (lam), (rec) and (fold)) and rules for eliminating types (the destructors (case), (fst), (snd), (app) and (unfold)). The remaining rules (var) and (weak) only work on the context and therefore do not fit in these categories.

Renaming convention for terms. In these rules case binds the variable x in the term l and y in r, λ binds x in t and rec binds g in t. Like we have done with types we do not distinguish between terms that are equal except that the names of their bound variables differ.

Notation. Similar to the type notation $\sigma(\vec{X})$ we write t^σ to express that t is of type σ, $t[\Gamma]$ that t is a term over context Γ and $t^\sigma[\Gamma]$ to express both, i.e. $t \in \mathrm{Tm}^\sigma[\Gamma]$. We define the set of *closed terms* Tm^σ of type σ as the set of terms over an empty context $\mathrm{Tm}^\sigma[]$.

2.3 Examples

To strengthen the reader's understanding of our language we will present a few examples. By the way we see that we can embed Gödel's T and Kleene's $\mathcal{O}$ into our system. Identifiers for defined expressions of our language are printed in **bold** font.

2.3.1 Natural numbers

We define the natural numbers as the recursive type

$$\mathbf{Nat} \equiv \mathrm{Rec}\, X.1 + X$$

and zero and successor as

$$\begin{aligned} \mathbf{O} &\equiv \mathrm{fold}(\mathrm{inl}()) \\ \mathbf{S}(v) &\equiv \mathrm{fold}(\mathrm{inr}(v)) \end{aligned}$$

Addition on **Nat** could be defined as the following function $\mathbf{add} \in \mathrm{Tm}^{\mathbf{Nat}\to\mathbf{Nat}\to\mathbf{Nat}}$:

$$\begin{aligned} \mathbf{add} \equiv\ & \mathrm{rec}\, \mathrm{add}^{\mathbf{Nat}\to\mathbf{Nat}\to\mathbf{Nat}}.\ \lambda x^{\mathbf{Nat}}.\ \lambda y^{\mathbf{Nat}}.\ \mathrm{case}(\mathrm{unfold}(x), \\ & \_^1.\ y, \\ & x'^{\mathbf{Nat}}.\ \mathbf{S}(\mathrm{add}\, x'\, y)) \end{aligned}$$

We use _ as variable name for a variable of type 1, since its content will always be () and thus we never have to refer to it.

We even can define the recursor $\mathbf{R}^\sigma$ over natural numbers for result type σ, that is a constant in Gödel's T:

$$\begin{aligned} \mathbf{R}^\sigma \equiv\ & \mathrm{rec}\, \mathrm{R}^{\sigma\to(\mathbf{Nat}\to\sigma\to\sigma)\to\mathbf{Nat}\to\sigma}.\ \lambda f_O^\sigma.\ \lambda f_S^{\mathbf{Nat}\to\sigma\to\sigma}.\ \lambda n^{\mathbf{Nat}}. \\ & \mathrm{case}(\mathrm{unfold}(n), \\ & \quad \_^1.\ f_O, \\ & \quad n'^{\mathbf{Nat}}.\ f_S\, n'\ (\mathrm{R}\, f_O\, f_S\, n')) \end{aligned}$$

2.3.2 Ordinal numbers

In our system we can define ordinal numbers as follows, where we represent limes numbers as functions from **Nat** to **Ord**:

$$\mathbf{Ord} \equiv \mathrm{Rec}\, X.(1 + X) + (\mathbf{Nat} \to X)$$

As above we can define zero, successor, limes and addition:

$$\begin{array}{rcl} \mathbf{O} & \equiv & \mathrm{fold}(\mathrm{inl}(\mathrm{inl}())) \\ \mathbf{S}(v) & \equiv & \mathrm{fold}(\mathrm{inl}(\mathrm{inr}(v))) \\ \mathbf{Lim}(f) & \equiv & \mathrm{fold}(\mathrm{inr}(f)) \end{array}$$

$$\begin{array}{rl} \mathbf{addOrd} \equiv & \mathrm{rec}\,\mathrm{addOrd}^{\mathbf{Ord}\to\mathbf{Ord}\to\mathbf{Ord}}.\ \lambda x^{\mathbf{Ord}}.\ \lambda y^{\mathbf{Ord}}.\ \mathrm{case}(\mathrm{unfold}(x), \\ & n^{1+\mathbf{Ord}}.\ \mathrm{case}(n, \\ & \qquad\qquad \_^{1}.\ y, \\ & \qquad\qquad x'^{\mathbf{Ord}}.\ \mathbf{S}(\mathrm{addOrd}\, x'\, y)) \\ & f^{\mathbf{Nat}\to\mathbf{Ord}}.\ \mathbf{Lim}(\lambda z^{\mathbf{Nat}}.\ \mathrm{addOrd}\ (f\ z)\ y)) \end{array}$$

We will extend these examples in the next sections.

3 Operational semantics

We only assign a meaning to closed terms $t \in \mathrm{Tm}^\sigma$: they evaluate to *(syntactic) values* of type σ. (We want to define *semantic* values as well, see section 4.) But during the process of evaluation defined by our operational semantics we will have to handle open terms, where their free variables are assigned values we store in an environment. Term and its corresponding environment form a *closure* which can be seen as completion of an open term.

3.1 Values and closures

In formulations of the lambda calculus in the literature (see for instance [Mat98] for an accurate formalization) often normal forms are defined as terms that cannot be reduced further.[4] Since we do not want open terms as results of an evaluation, we follow a different approach: we define values from scratch and later show that they correspond to closed terms resp. closures (see section 3.3).

Values. We define Val^σ inductively as follows. Again we write v^σ to express $v \in \mathrm{Val}^\sigma$. The set of environments $\mathrm{Val}(\Gamma)$ is defined simultaneously (see the definition below).

$$\text{(vlam)}\ \frac{t \in \mathrm{Tm}^\tau[\Gamma, x^\sigma] \quad e \in \mathrm{Val}(\Gamma)}{\langle \lambda x^\sigma.t; e\rangle \in \mathrm{Val}^{\sigma\to\tau}} \qquad \text{(vunit)}\ \frac{}{() \in \mathrm{Val}^1}$$

$$\text{(vrec)}\ \frac{t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma, g^{\sigma\to\tau}] \quad e \in \mathrm{Val}(\Gamma)}{\langle \mathrm{rec}\, g^{\sigma\to\tau}.t; e\rangle \in \mathrm{Val}^{\sigma\to\tau}}$$

$$\text{(vinl)}\ \frac{v \in \mathrm{Val}^\sigma \quad \tau \in \mathrm{Ty}}{\mathrm{inl}(v) \in \mathrm{Val}^{\sigma+\tau}} \qquad \text{(vinr)}\ \frac{v \in \mathrm{Val}^\tau \quad \sigma \in \mathrm{Ty}}{\mathrm{inr}(v) \in \mathrm{Val}^{\sigma+\tau}}$$

$$\text{(vpair)}\ \frac{v \in \mathrm{Val}^\sigma \quad w \in \mathrm{Val}^\tau}{(v, w) \in \mathrm{Val}^{\sigma\times\tau}} \qquad \text{(vfold)}\ \frac{v \in \mathrm{Val}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}}{\mathrm{fold}(v) \in \mathrm{Val}^{\mathrm{Rec}\, X.\sigma(X)}}$$

These rules correspond to the introduction rules for terms. Since values represent evaluated terms, we need only constructors, no destructors.

[4] Ralph Matthes first defines normal forms inductively and later proves that they are the irreducible terms.

Environments and Closures. We define the *set of environments* of the context $\Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ as

$$\mathrm{Val}(\Gamma) := \{x_1 = v_1, \ldots, x_n = v_n : v_i \in \mathrm{Val}^{\sigma_i}\}$$

We write e^Γ for an environment $e \in \mathrm{Val}(\Gamma)$ and "$\cdot$" for the empty environment. The *set of closures* of type τ we define as

$$\begin{array}{rcl} \mathrm{Cl}^\tau & := & \{\langle t; e\rangle : \Gamma \in \mathrm{Cxt}, t \in \mathrm{Tm}^\tau[\Gamma], e \in \mathrm{Val}(\Gamma)\} \\ & \cup & \{f@u : f \in \mathrm{Val}^{\sigma\to\tau}, u \in \mathrm{Val}^\sigma\} \end{array}$$

Here @ is a syntactic function symbol $\mathrm{Val}^{\sigma\to\tau} \times \mathrm{Val}^\sigma \to \mathrm{Cl}^\tau$. Closures of the form $f@u$ (value applied to values) are convenient to define the operational semantics without typecasting values back to terms, which in an implementation would be inefficient as well.

Renaming convention for closures. Since in a closure all variables of a term are bound, we consider two closures as equal that differ only in variable names.

3.2 Operational semantics

In the following we present a big step operational semantics "$\Downarrow$" that defines how closures are evaluated to values. Our strategy is call-by-value (see rule (opapp)) and we do not evaluate under λ and rec (see rules (oplam) and (oprec). Furthermore it is deterministic, i.e. for every closure there is at most one computation tree.[5]

Operational semantics. We inductively define a family of relations $\Downarrow^\sigma \subseteq \mathrm{Cl}^\sigma \times \mathrm{Val}^\sigma$ indexed over Ty. As the type σ can be inferred from the type of the closure or the value, we generally leave it out. For reasons of readability we leave out type and context annotations wherever possible.

[5]For all closures except $\langle \mathrm{case}(t, x.l, y.r); \ldots\rangle$ there is only one computation rule. But also for closures with case analysis there will be only one computation tree, since $\langle t; \ldots\rangle$ evaluates to either a left (inl) or a right (inr) injection of some value and thus only one of the rules (opcasel) or (opcaser) is applicable.

$$\text{(opunit)} \quad \frac{}{\langle (); \cdot \rangle \Downarrow ()}$$

$$\text{(opvar)} \quad \frac{}{\langle x; e, x = v \rangle \Downarrow v} \qquad \text{(opweak)} \quad \frac{\langle t[\Gamma]; e \rangle \Downarrow v}{\langle t[\Gamma, x]; e, x = w \rangle \Downarrow v}$$

$$\text{(opinl)} \quad \frac{\langle t; e \rangle \Downarrow v}{\langle \mathrm{inl}(t); e \rangle \Downarrow \mathrm{inl}(v)} \qquad \text{(opinr)} \quad \frac{\langle t; e \rangle \Downarrow v}{\langle \mathrm{inr}(t); e \rangle \Downarrow \mathrm{inr}(v)}$$

$$\text{(opcasel)} \quad \frac{\langle t^{\sigma+\tau}[\Gamma]; e \rangle \Downarrow \mathrm{inl}(w^{\sigma}) \quad \langle l^{\rho}[\Gamma, x^{\sigma}]; e, x = w \rangle \Downarrow v^{\rho}}{\langle \mathrm{case}(t, x.l, y.r); e \rangle \Downarrow v}$$

$$\text{(opcaser)} \quad \frac{\langle t^{\sigma+\tau}[\Gamma]; e \rangle \Downarrow \mathrm{inr}(w^{\tau}) \quad \langle r^{\rho}[\Gamma, y^{\tau}]; e, x = w \rangle \Downarrow v^{\rho}}{\langle \mathrm{case}(t, x.l, y.r); e \rangle \Downarrow v}$$

$$\text{(oppair)} \quad \frac{\langle s; e \rangle \Downarrow v \quad \langle t; e \rangle \Downarrow w}{\langle (s, t); e \rangle \Downarrow (v, w)}$$

$$\text{(opfst)} \quad \frac{\langle p; e \rangle \Downarrow (v, w)}{\langle \mathrm{fst}(p); e \rangle \Downarrow v} \qquad \text{(opsnd)} \quad \frac{\langle p; e \rangle \Downarrow (v, w)}{\langle \mathrm{snd}(p); e \rangle \Downarrow w}$$

$$\text{(oplam)} \quad \frac{}{\langle \lambda x.t; e \rangle \Downarrow \langle \lambda x.t; e \rangle} \qquad \text{(oprec)} \quad \frac{}{\langle \mathrm{rec}\, g.t; e \rangle \Downarrow \langle \mathrm{rec}\, g.t; e \rangle}$$

$$\text{(opapp)} \quad \frac{\langle t; e \rangle \Downarrow f \quad \langle s; e \rangle \Downarrow u \quad f@u \Downarrow v}{\langle t\, s; e \rangle \Downarrow v}$$

$$\text{(opappvl)} \quad \frac{\langle t; e, x = u \rangle \Downarrow v}{\langle \lambda x.t; e \rangle @u \Downarrow v}$$

$$\text{(opappvr)} \quad \frac{\langle t; e, g = \mathrm{rec}\, g.t \rangle \Downarrow f \quad f@u \Downarrow v}{\langle \mathrm{rec}\, g.t; e \rangle @u \Downarrow v}$$

$$\text{(opfold)} \quad \frac{\langle t; e \rangle \Downarrow v}{\langle \mathrm{fold}(t); e \rangle \Downarrow \mathrm{fold}(v)} \qquad \text{(opunfold)} \quad \frac{\langle t; e \rangle \Downarrow \mathrm{fold}(v)}{\langle \mathrm{unfold}(t); e \rangle \Downarrow v}$$

3.3 Correspondence of values and closures

For our operational semantics we want to show a consistency property: Values, converted back to closures, should evaluate to themselves. To prove this proposition we need some

Additional weakening rules. We specify two rules for terms that perform weakening by introduction of several variables at a time: By iterated application of (weak) we can weaken a term by a list of variables Δ:

$$(\text{weakEnd}) \qquad \frac{t \in \mathrm{Tm}^\sigma[\Gamma] \quad \Gamma \cap \Delta = \emptyset}{t \in \mathrm{Tm}^\sigma[\Gamma, \Delta]}$$

We also need to be able to weaken a term by adding variables at the beginning of its context.

$$(\text{weakBeg}) \qquad \frac{t \in \mathrm{Tm}^\sigma[\Gamma] \quad \Gamma \cap \Delta = \emptyset}{t \in \mathrm{Tm}^\sigma[\Delta, \Gamma]}$$

That rule we get by decomposing the term into its derivation tree, weakening the context at the leaves[6] of this tree and then re-composing the term. In the same way we get evaluation rules for weakened closures:

$$(\text{opweakEnd}) \qquad \frac{\langle t[\Gamma]; e^\Gamma \rangle \Downarrow v}{\langle t[\Gamma, \Delta]; e^\Gamma, d^\Delta \rangle \Downarrow v}$$

$$(\text{opweakBeg}) \qquad \frac{\langle t[\Gamma]; e^\Gamma \rangle \Downarrow v}{\langle t[\Delta, \Gamma]; d^\Delta, e^\Gamma \rangle \Downarrow v}$$

Consistency. To prove that each value evaluates to itself, we must convert it back to its closure first. To this end, we define the canonical embedding

$$\mathrm{cl}^\sigma : \mathrm{Val}^\sigma \hookrightarrow \mathrm{Cl}^\sigma$$

by recursion on values as follows:

[6]There are two kinds of leaves:

(var) Here we can weaken the context in the beginning, since we can introduce a new variable with any context.

(unit) Since here the context is empty, we can insert the necessary weakening step.

(vunit) $\mathrm{cl}() := \langle (); \cdot \rangle$

(vlam) $\mathrm{cl}\langle \lambda x.t; e\rangle := \langle \lambda x.t; e\rangle$

(vrec) $\mathrm{cl}\langle \mathrm{rec}\, g.t; e\rangle := \langle \mathrm{rec}\, g.t; e\rangle$

(vinl) Let $\langle t; e\rangle := \mathrm{cl}(v)$. Then $\mathrm{cl}(\mathrm{inl}(v)) := \langle \mathrm{inl}(t); e\rangle$

(vinr) Let $\langle t; e\rangle := \mathrm{cl}(v)$. Then $\mathrm{cl}(\mathrm{inr}(v)) := \langle \mathrm{inr}(t); e\rangle$

(vpair) Let $\langle s[\Delta]; d\rangle := \mathrm{cl}(v)$, $\langle t[\Gamma]; e\rangle := \mathrm{cl}(w)$. By the renaming convention for closures we can assume that $\Gamma \cap \Delta = \emptyset$. We weaken $s[\Delta]$ to $s[\Delta, \Gamma]$ by weakEnd and $t[\Gamma]$ to $t[\Delta, \Gamma]$ by weakBeg. Then $\mathrm{cl}(v, w) := \langle (s, t)[\Delta, \Gamma]; d, e\rangle$

(vfold) Let $\langle t; e\rangle := \mathrm{cl}(v)$. Then $\mathrm{cl}(\mathrm{fold}(v)) := \langle \mathrm{fold}(t); e\rangle$

Proposition 1 (Values evaluate to themselves)

$$\forall \sigma \in \mathrm{Ty}, v \in \mathrm{Val}^{\sigma}.\ \mathrm{cl}(v) \Downarrow v$$

Proof by induction over v. The base cases (vunit), (vlam) and (vrec) are trivial by (opunit), (oplam) and (oprec).

(vinl) By induction hypothesis (IH) we have $\mathrm{cl}(v) =: \langle t; e\rangle \Downarrow v$. Using (opinl) we get $\mathrm{cl}(\mathrm{inl}(v)) = \langle \mathrm{inl}(t); e\rangle \Downarrow \mathrm{inl}(v)$.

(vinr) analogously

(vfold) analogously by (opfold)

(vpair) By IH we have $\mathrm{cl}(v) =: \langle s; d\rangle \Downarrow v$. Using (opweakEnd) we get $\langle s; d, e\rangle \Downarrow v$. Also by IH we have $\mathrm{cl}(w) =: \langle t; e\rangle \Downarrow w$, hence by (opweakBeg) $\langle t; d, e\rangle \Downarrow w$. (oppair) finishes the proof producing $\mathrm{cl}(v, w) = \langle (s, t); d, e\rangle \Downarrow (v, w)$ □

3.4 Examples

In the following we will show two properties of the type **Nat** defined in section 2.3.1.

3.4.1 Numerals

Recall the definition of $\mathbf{Nat} \equiv \mathrm{Rec}\, X.1 + X$, $\mathbf{O}$ and $\mathbf{S}$ in section 2.3.1. We can show that the numerals $\mathrm{Val}^{\mathbf{Nat}}$ are isomorphic to the natural numbers $\mathbb{N}$.

Proposition 2

$$\mathrm{Val}^{\mathbf{Nat}} = \{\mathbf{S}^n(\mathbf{O}) : n \in \mathbb{N}\}$$

Proof. For "$\supseteq$" we show $\forall n \in \mathbb{N}.\ \mathbf{S}^n(\mathbf{O}) \in \mathrm{Val}^{\mathbf{Nat}}$ by induction over n.

$$n = 0 \qquad \dfrac{\dfrac{\dfrac{}{() \in \mathrm{Val}^1}\,\text{vunit}}{\mathrm{inl}() \in \mathrm{Val}^{1+\mathbf{Nat}}}\,\text{vinl}}{\mathbf{O} \equiv \mathrm{fold}(\mathrm{inl}()) \in \mathrm{Val}^{\mathbf{Nat}}}\,\text{vfold}$$

$$n \to n{+}1 \qquad \dfrac{\dfrac{\dfrac{}{\mathbf{S}^n(\mathbf{O}) \in \mathrm{Val}^{\mathbf{Nat}}}\,\text{IH}}{\mathrm{inr}(\mathbf{S}^n(\mathbf{O})) \in \mathrm{Val}^{1+\mathbf{Nat}}}\,\text{vinr}}{\mathbf{S}^{n+1}(\mathbf{O}) \equiv \mathrm{fold}(\mathrm{inr}(\mathbf{S}^n(\mathbf{O}))) \in \mathrm{Val}^{\mathbf{Nat}}}\,\text{vfold}$$

The other direction "$\subseteq$": We show simultaneously

$$\begin{array}{ll} \text{(i)} & v \in \mathrm{Val}^{\mathbf{Nat}} \to v \in \{\mathbf{S}^n(\mathbf{O}) : n \in \mathbb{N}\} \\ \text{(ii)} & v \in \mathrm{Val}^{1+\mathbf{Nat}} \to v \in \{\mathrm{inl}(), \mathrm{inr}(\mathbf{S}^n(\mathbf{O})) : n \in \mathbb{N}\} \end{array}$$

by induction on generation of v. The only matching cases are:

(vinl) (ii) $v \equiv \mathrm{inl}() \in \mathrm{Val}^{1+\mathbf{Nat}}$: Obviously $v \in \{\mathrm{inl}(), \mathrm{inr}(\mathbf{S}^n(\mathbf{O})) : n \in \mathbb{N}\}$.

(vinr) (ii) $v \equiv \mathrm{inr}(v')$, $v' \in \mathrm{Val}^{\mathbf{Nat}}$: By induction hypothesis (i) we have $v' \in \{\mathbf{S}^n(\mathbf{O}) : n \in \mathbb{N}\}$, therefore we see immediately that $v \in \{\mathrm{inl}(), \mathrm{inr}(\mathbf{S}^n(\mathbf{O})) : n \in \mathbb{N}\}$.

(vfold) (i) $v \equiv \mathrm{fold}(v')$, $v' \in \mathrm{Val}^{1+\mathbf{Nat}}$: By induction hypothesis (ii) either $v' = \mathrm{inl}()$ or there is an $n \in \mathbb{N}$ with $v' = \mathrm{inr}(\mathbf{S}^n(\mathbf{O}))$. In the first case $v = \mathbf{O}$ and in the second case $v = \mathbf{S}^{n+1}(\mathbf{O})$, thus in both cases $v \in \{\mathbf{S}^n(\mathbf{O}) : n \in \mathbb{N}\}$. $\square$

For the following we introduce the abbreviation $\overline{n} := \mathbf{S}^n(\mathbf{O})$ for numerals.

3.4.2 Soundness of addition

In order to show that our function **add** really implements addition on numbers, we have to prove

$$\forall n, m \in \mathbb{N}.\ \langle \mathbf{add}\ \mathbf{S}^n(\mathbf{O})\ \mathbf{S}^m(\mathbf{O}); \cdot\rangle \Downarrow \mathbf{S}^{n+m}(\mathbf{O})$$

Here we want to show it for the *values* $\langle \mathbf{add}; \cdot\rangle \in \mathrm{Val}^{\mathbf{Nat}\to\mathbf{Nat}\to\mathbf{Nat}}$, $\overline{n}, \overline{m} \in \mathrm{Val}^{\mathbf{Nat}}$. (In the following we will write **add** also for its value $\langle \mathbf{add}; \cdot\rangle$.) However, $(\mathbf{add}@\overline{n})@\overline{m}$ is not syntactically correct, because the application @ is a function symbol $\mathrm{Val}^{\sigma\to\tau} \times \mathrm{Val}^{\sigma} \to \mathrm{Cl}^{\tau}$. Therefore we need a function $\mathbf{add}_{\overline{n}} \in \mathrm{Val}^{\mathbf{Nat}\to\mathbf{Nat}}$ for each numeral $\overline{n} \in \mathrm{Val}^{\mathbf{Nat}}$, which we define as the result of evaluating $\mathbf{add}@\overline{n}$, i.e.

$$\mathbf{add}_{\overline{n}} = \varepsilon v.\, \mathbf{add}@\overline{n} \Downarrow v$$

(Hilbert's epsilon operator $\varepsilon x.\ P(x)$ picks an arbitrary element with the postulated property P; that implies $P(\varepsilon x.\ P(x))$ is true.) We can compute $\mathbf{add}_{\overline{n}}$ by our operational semantics:

$$\genfrac{}{}{0pt}{}{\dfrac{\dfrac{}{\langle \lambda x \ldots\rangle \Downarrow \langle \lambda x \ldots\rangle}\,\mathrm{oplam} \qquad \dfrac{\dfrac{}{\langle \lambda y \ldots\rangle \Downarrow \langle \lambda y \ldots\rangle}\,\mathrm{oplam}}{\langle \lambda x.\, \lambda y \ldots; \mathrm{add} = \mathbf{add}\rangle@\overline{n} \Downarrow \langle \lambda y \ldots, x = \overline{n}\rangle}\,\mathrm{opappvl}}{\langle \mathrm{rec\ add}.\lambda x.\, \lambda y.\, \mathrm{case}(\ldots); \cdot\rangle@\overline{n} \Downarrow \langle \lambda y.\, \mathrm{case}(\ldots); \mathrm{add} = \mathbf{add}, x = \overline{n}\rangle}\,\mathrm{opappvr}}{\overline{\overline{\mathbf{add}@\overline{n} \Downarrow \mathbf{add}_{\overline{n}}}}\ \mathrm{def}}$$

Now we can prove the correctness of **add**, i.e. we show

Proposition 3

$$\forall n, m \in \mathbb{N}.\, \mathbf{add}_{\overline{n}}@\overline{m} \Downarrow \overline{n+m}$$

Proof by induction over n. Case $n = 0$: We show $\mathbf{add}_{\mathbf{O}}@\overline{m} \Downarrow \overline{m}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x; \ldots x = \mathbf{O}\rangle \Downarrow \mathbf{O} \equiv \mathrm{fold}(\mathrm{inl}())}\,\mathrm{opvar}}{\langle \mathrm{unfold}(x); \ldots x = \mathbf{O}\rangle \Downarrow \mathrm{inl}()}\,\mathrm{opunfold}}{\langle \mathrm{unfold}(x); \ldots x = \mathbf{O}, y = \overline{m}\rangle \Downarrow \mathrm{inl}()}\,\mathrm{opweak} \qquad \dfrac{\dfrac{}{\langle y; \ldots y = \overline{m}\rangle \Downarrow \overline{m}}\,\mathrm{opvar}}{\langle y; \ldots y = \overline{m}, \_ = ()\rangle \Downarrow \overline{m}}\,\mathrm{opweak}}{\langle \mathrm{case}(\mathrm{unfold}(x), \_^{1}.y, x'^{\mathbf{Nat}}.\mathbf{S}(\mathrm{add}\ x'\ y); \mathrm{add} = \mathbf{add}, x = \mathbf{O}, y = \overline{m}\rangle \Downarrow \overline{m}}\,\mathrm{opcasel}}{\langle \lambda y^{\mathbf{Nat}}.\, \mathrm{case}(\ldots); \mathrm{add} = \mathbf{add}, x = \mathbf{O}\rangle@\overline{m} \Downarrow \overline{m}}\,\mathrm{opappvl}$$

Case $n \to n{+}1$: Given the induction hypothesis $\mathbf{add}_{\overline{n}}@\overline{m} \Downarrow \overline{n+m}$. we show $\mathbf{add}_{\mathbf{S}(\overline{n})}@\overline{m} \Downarrow \mathbf{S}(\overline{n+m})$. For technical reasons we will split the proof tree into three parts:

(I) Derivation of $\langle \mathrm{add}\, x'; \ldots x' = \overline{n}\rangle \Downarrow \mathbf{add}_{\overline{n}}$:

$$\dfrac{\begin{array}{c}\vdots\\ \langle \mathrm{add}; \ldots\rangle \Downarrow \mathbf{add}\end{array} \quad \begin{array}{c}\vdots\\ \langle x'; \ldots x' = \overline{n}\rangle \Downarrow \overline{n}\end{array} \quad \begin{array}{c}\vdots\ \text{Def. of } \mathbf{add}_{\overline{n}}\\ \mathbf{add}@\overline{n} \Downarrow \mathbf{add}_{\overline{n}}\end{array}}{\langle \mathrm{add}\, x'; \ldots x' = \overline{n}\rangle \Downarrow \mathbf{add}_{\overline{n}}}\ \text{opapp}$$

(II) Derivation of $\langle \mathbf{S}(\mathrm{add}\, x'\, y); \ldots y = \overline{m}, x' = \overline{n}\rangle \Downarrow \mathbf{S}(\overline{n+m})$:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\ (\mathrm{I})\\ \langle \mathrm{add}\, x'; \ldots x' = \overline{n}\rangle \Downarrow \mathbf{add}_{\overline{n}}\end{array} \quad \begin{array}{c}\vdots\\ \langle y; \ldots\rangle \Downarrow \overline{m}\end{array} \quad \dfrac{}{\mathbf{add}_{\overline{n}}@\overline{m} \Downarrow \overline{n+m}}\ \text{IH}}{\langle (\mathrm{add}\, x')\, y; \ldots y = \overline{m}, x' = \overline{n}\rangle \Downarrow \overline{n+m}}\ \text{opapp}}{\langle \mathrm{inr}(\mathrm{add}\, x'\, y); \ldots y = \overline{m}, x' = \overline{n}\rangle \Downarrow \mathrm{inr}(\overline{n+m})}\ \text{opinr}}{\langle \mathbf{S}(\mathrm{add}\, x'\, y); \ldots y = \overline{m}, x' = \overline{n}\rangle \Downarrow \mathbf{S}(\overline{n+m})}\ \text{opfold}$$

(III) Derivation of $\mathbf{add}_{\mathbf{S}(\overline{n})}@\overline{m} \Downarrow \mathbf{S}(\overline{n+m})$:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \langle \mathrm{unfold}(x); \ldots x{=}\mathbf{S}(\overline{n})\ldots\rangle \Downarrow \mathrm{inr}(\overline{n})\end{array} \quad \begin{array}{c}\vdots\ (\mathrm{II})\\ \langle \mathbf{S}(\mathrm{add}\, x'\, y); \ldots y{=}\overline{m}, x'{=}\overline{n}\rangle \Downarrow \mathbf{S}(\overline{n+m})\end{array}}{\langle \mathrm{case}(\mathrm{unfold}(x), \_^{1}.y, x'^{\mathbf{Nat}}.\mathbf{S}(\mathrm{add}\, x'\, y)); \ldots x = \mathbf{S}(\overline{n}), y = \overline{m}\rangle \Downarrow \mathbf{S}(\overline{n+m})}\ \text{opcaser}}{\langle \lambda y^{\mathbf{Nat}}.\, \mathrm{case}(\ldots); \mathrm{add} = \mathbf{add}, x = \mathbf{S}(\overline{n})\rangle @\overline{m} \Downarrow \mathbf{S}(\overline{n+m})}\ \text{opappvl}}{\mathbf{add}_{\mathbf{S}(\overline{n})}@\overline{m} \Downarrow \mathbf{S}(\overline{n+m})}\ \text{def}$$

□

4 Semantic values

Now that we have built a partial language by defining types, terms and an operational semantics, we are interested in a sublanguage that is total. Therefore we define a semantics $[\![\cdot]\!]$ over the types, so that $[\![\sigma]\!]$ contains all "good" values of type σ, i.e. all values that guarantee termination.

Since recursive types $\mathrm{Rec}\, X.\sigma(X)$ are interpreted as the least fixed point of the operator represented by $\sigma(X)$, we have to ensure monotonicity of our types. In a lemma we will show that all types are monotone since we have restricted ourselves to strictly positive types. But first some facts about the

Least fixed point. Let $(\mathcal{U}, \subseteq)$ be a complete lattice[7] of sets. By Tarski's fixed-point theorem [Tar55] every monotone[8] operator $\mathcal{F} : \mathcal{U} \to \mathcal{U}$ has a least fixed point $F := \mathrm{lfp}\, \mathcal{F}$ with the following two properties

$$\begin{array}{rl} \text{(ispfp)} & \mathcal{F}(F) \subseteq F \\ \text{(ismpfp)} & \forall A \in \mathcal{U}.\, \mathcal{F}(A) \subseteq A \to F \subseteq A \end{array}$$

which characterize F as the least *pre*-fixed point of $\mathcal{F}$. We can show that

$$F = \inf \{A \in \mathcal{U} : \mathcal{F}(A) \subseteq A\}$$

We get "$\supseteq$" by (ispfp) since $F \in \mathcal{A} := \{A \in \mathcal{U} : \mathcal{F}(A) \subseteq A\}$ and "$\subseteq$" by (ismpfp) immediately.

For F to be the fixed point we need $F \subseteq \mathcal{F}(F)$. We obtain this as follows: Since $\mathcal{F}$ is monotone $\mathcal{F}(\mathcal{F}(F)) \subseteq \mathcal{F}(F)$ by (ispfp), hence $\mathcal{F}(F) \in \mathcal{A}$ and therefore by (ismpfp) $F \subseteq \mathcal{F}(F)$.

Folding of value sets. For convenience we define a polymorphic bijective function fold that performs folding on all elements of a value set:

$$\begin{array}{rcl} \mathrm{fold} & : & \mathcal{P}\left(\mathrm{Val}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}\right) \to \mathcal{P}\left(\mathrm{Val}^{\mathrm{Rec}\, X.\sigma(X))}\right) \\ & & W \mapsto \{\mathrm{fold}(v) : v \in W\} \end{array}$$

From the rule (vfold) and its inversion we can derive that fold is monotone.

[7] i.e. to every collection $\mathcal{A} \subseteq \mathcal{U}$ of sets there is an infimum $\inf \mathcal{A}$ w.r.t. set inclusion "$\subseteq$".

[8] $\mathcal{F}$ monotone $:\leftrightarrow \forall A \subseteq B.\, \mathcal{F}(A) \subseteq \mathcal{F}(B)$.

Semantic Values. The idea is that good values of $\to$-type are the functions f that for every good input u produce a good output v. (This of course implies termination of f on all good inputs.) Good values of other types have no other restriction, except that they be derived from good values of the component types.

Let $\sigma(\vec{X})$ be a type over the free type variables $\vec{X} = X_1, \dots, X_n$; $\vec{\tau} = \tau_1, \dots, \tau_n$ a list of closed types and $\vec{V} = V_1, \dots, V_n \subseteq \mathrm{Val}^{\vec{\tau}}$ sets of (syntactic) values of type $\vec{\tau}$ (i.e. $V_1 \subseteq \mathrm{Val}^{\tau_1}, \dots, V_n \subseteq \mathrm{Val}^{\tau_n}$). The set of semantic values $[\![\sigma(\vec{X})]\!]_{\vec{V}} \subseteq \mathrm{Val}^{\sigma(\vec{\tau})}$ of type σ over the value sets $\vec{V}$ is a subset of the values of the type σ the variables of which are substituted by $\vec{\tau}$. It is defined by recursion over σ as follows:

(Unit) $[\![1]\!] := \{()\}$

(Var) $[\![X_n]\!]_{\vec{V}} := V_n$

(Weak) $[\![\sigma(\vec{X}, Y)]\!]_{\vec{V},W} := [\![\sigma(\vec{X})]\!]_{\vec{V}}$

(Sum) $[\![(\sigma + \tau)(\vec{X})]\!]_{\vec{V}} := \{\mathrm{inl}(v) : v \in [\![\sigma(\vec{X})]\!]_{\vec{V}}\} \cup \{\mathrm{inr}(v) : v \in [\![\tau(\vec{X})]\!]_{\vec{V}}\}$

(Prod) $[\![(\sigma \times \tau)(\vec{X})]\!]_{\vec{V}} := \{(v, w) : v \in [\![\sigma(\vec{X})]\!]_{\vec{V}}, w \in [\![\tau(\vec{X})]\!]_{\vec{V}}\}$

(Arr) $[\![\sigma \to \tau(\vec{X})]\!]_{\vec{V}} := \{f \in \mathrm{Val}^{\sigma \to \tau(\vec{\tau})} : \forall u \in [\![\sigma]\!].\ \exists v \in [\![\tau(\vec{X})]\!]_{\vec{V}}.\ f@u \Downarrow v\}$

(Rec) $[\![\mathrm{Rec}\, Y.\sigma(\vec{X}, Y)]\!]_{\vec{V}} := \mathrm{lfp}\, \mathcal{F}$, where we define $\mathcal{F}$ as

$$\begin{aligned} \mathcal{F} \ :\ & \mathcal{P}\left(\mathrm{Val}^{\mathrm{Rec}\, Y.\sigma(\vec{\tau}, Y)}\right) \to \mathcal{P}\left(\mathrm{Val}^{\mathrm{Rec}\, Y.\sigma(\vec{\tau}, Y)}\right) \\ & W \mapsto \mathrm{fold}\left([\![\sigma(\vec{X}, Y)]\!]_{\vec{V},W}\right) \end{aligned}$$

For this definition to be valid the least fixed point in (Rec) must exist. This is only guaranteed if the operator $\mathcal{F}$ is monotone, which we show simultaneously to the definition. For reasons of clarity we have written it down separately in the following

Lemma 4 (Monotonicity of semantic values)

$$\forall \sigma(\vec{X}, Y, \vec{Z}).\ A \subseteq B \to [\![\sigma(\vec{X}, Y, \vec{Z})]\!]_{\vec{V},A,\vec{W}} \subseteq [\![\sigma(\vec{X}, Y, \vec{Z})]\!]_{\vec{V},B,\vec{W}}$$

Proof by induction on σ. Here the strict positivity of σ comes in decisively (see case (Arr)).

(Unit) Nothing to show.

(Var) $[\![X_n]\!]_{\vec{V}} = V_n \subseteq V_n = [\![X_n]\!]_{\vec{V}}$
$[\![Y]\!]_{\vec{V},A} = A \subseteq B = [\![Y]\!]_{\vec{V},A}$
$[\![Z_n]\!]_{\vec{V},A,\vec{W}} = W_n \subseteq W_n = [\![Z_i]\!]_{\vec{V},A,\vec{W}}$

(Weak) by Y: $[\![\sigma(\vec{X},Y)]\!]_{\vec{V},A} = [\![\sigma(\vec{X})]\!]_{\vec{V}} = [\![\sigma(\vec{X},Y)]\!]_{\vec{V},B}$
by Z: $[\![\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},A,\vec{W},W} = [\![\sigma(\vec{X},Y,\vec{Z})]\!]_{\vec{V},A,\vec{W}} \subseteq$
$[\![\sigma(\vec{X},$
$Y,\vec{Z})]\!]_{\vec{V},B,\vec{W}} = [\![\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},B,\vec{W},W}$ by IH.

Now for reasons of readability we leave out $\vec{X}$, $\vec{Z}$, $\vec{V}$ and $\vec{W}$.

(Sum) By IH we have $[\![\sigma(Y)]\!]_A \subseteq [\![\sigma(Y)]\!]_B$ and $[\![\tau(Y)]\!]_A \subseteq [\![\tau(Y)]\!]_B$. Thus $[\![(\sigma+\tau)(Y)]\!]_A = \underbrace{\{\mathrm{inl}(v) : v \in [\![\sigma(Y)]\!]_A\}}_{\subseteq\{\mathrm{inl}(v):v\in[\![\sigma(Y)]\!]_B\}} \cup$
$\underbrace{\{\mathrm{inr}(v) : v \in [\![\tau(Y)]\!]_A\}}_{\subseteq\{\mathrm{inr}(v):v\in[\![\tau(Y)]\!]_B\}} \subseteq [\![(\sigma+\tau)(Y)]\!]_B$ by monotonicity of set union "$\cup$".

(Prod) $[\![\sigma(Y)]\!]_A \subseteq [\![\sigma(Y)]\!]_B$ and $[\![\tau(Y)]\!]_A \subseteq [\![\tau(Y)]\!]_B$ by IH. $[\![(\sigma \times \tau)(Y)]\!]_A = \{(v,w) : v \in [\![\sigma(Y)]\!]_A, w \in [\![\tau(Y)]\!]_A\} \subseteq \{(v,w) : v \in [\![\sigma(Y)]\!]_B, w \in [\![\tau(Y)]\!]_B\} = [\![(\sigma\times\tau)(Y)]\!]_B$ by monotonicity of "$\times$".

(Arr) Assume $f \in [\![\sigma \to \tau(Y)]\!]_A$ and $u \in [\![\sigma]\!]$. By definition there is a $v \in [\![\tau(Y)]\!]_A$ with $f@u \Downarrow v$. By IH $v \in [\![\tau(Y)]\!]_B$ and hence $f \in [\![\sigma \to \tau(Y)]\!]_B$.

(Rec) $[\![\mathrm{Rec}\, Z.\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},A,\vec{W}} \subseteq [\![\mathrm{Rec}\, Z.\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},B,\vec{W}}$ is what we have to show. We introduce the abbreviation $\mathcal{F}_{\_}(W) := \mathrm{fold}\left([\![\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},\_,\vec{W},W}\right)$ and thus or goal becomes $\mathrm{lfp}\,\mathcal{F}_A \subseteq \mathrm{lfp}\,\mathcal{F}_B$. By induction hypothesis we have $[\![\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},A,\vec{W},W} \subseteq [\![\sigma(\vec{X},Y,\vec{Z},Z)]\!]_{\vec{V},B,\vec{W},W}$ for all W and hence by monotonicity of fold $\mathcal{F}_A(\mathrm{lfp}\,\mathcal{F}_B) \subseteq \mathcal{F}_B(\mathrm{lfp}\,\mathcal{F}_B)$ for $W \equiv \mathrm{lfp}\,\mathcal{F}_B$. By (ispfp) for $\mathcal{F}_B$ we obtain $\mathcal{F}_A(\mathrm{lfp}\,\mathcal{F}_B) \subseteq \mathrm{lfp}\,\mathcal{F}_B$. Now we apply (ismpfp) for $\mathcal{F}_A$ and get $\mathrm{lfp}\,\mathcal{F}_A \subseteq \mathrm{lfp}\,\mathcal{F}_B$. □

Corollary 5

$$\vec{V} \subseteq \vec{W} \to [\![\sigma(\vec{X})]\!]_{\vec{V}} \subseteq [\![\sigma(\vec{X})]\!]_{\vec{W}}$$

Proof by iterated application of lemma 4.

For the desired monotonicity property $[\![\sigma(\vec{X})]\!]_{\vec{V}} \subseteq [\![\sigma(\vec{\tau})]\!]$ ($\vec{V} \subseteq [\![\vec{\tau}]\!]$) we need the corollary of the following

Lemma 6 (Substitution in semantic values)

$$[\![\sigma(\vec{X}, Y, \vec{Z})]\!]_{\vec{V},[\![\tau]\!],\vec{W}} = [\![\sigma(\vec{X}, \tau, \vec{Z})]\!]_{\vec{V},\vec{W}}$$

Proof by induction on σ. The case (Unit) is trivial, (Weak), (Sum), (Prod) and (Arr) are immediately shown by the induction hypothesis, so let us have a look at the remaining two:

(Var) $[\![Y]\!]_{\vec{V},[\![\tau]\!],\vec{W}} = [\![\tau]\!] = [\![Y[Y := \tau]]\!]_{\vec{V},\vec{W}}$

(Rec) We introduce the abbreviations $\mathcal{F}(W) := [\![\sigma(\vec{X}, Y, \vec{Z}, Z)]\!]_{\vec{V},[\![\tau]\!],\vec{W},W}$ and $\mathcal{G}(W) := [\![\sigma(\vec{X}, \tau, \vec{Z}, Z)]\!]_{\vec{V},\vec{W},W}$ and hence have to show lfp (fold $\circ\, \mathcal{F}$) = lfp (fold $\circ\, \mathcal{G}$). By IH we have $\forall W.\mathcal{F}(W) = \mathcal{G}(W)$, thus $\mathcal{F} = \mathcal{G}$ and also fold $\circ\, \mathcal{F}$ = fold $\circ\, \mathcal{G}$. Since the operators are equal, the fixed points are equal as well, hence lfp (fold $\circ\, \mathcal{F}$) = lfp (fold $\circ\, \mathcal{G}$). □

Corollary 7

$$[\![\sigma(\vec{X})]\!]_{[\![\vec{\tau}]\!]} = [\![\sigma(\vec{\tau})]\!]$$

Proof by iterated application of lemma 6.

Corollary 8 (Subset property of semantic values) *For all $\vec{V} \subseteq [\![\vec{\tau}]\!]$ we have*

$$[\![\sigma(\vec{X})]\!]_{\vec{V}} \subseteq [\![\sigma(\vec{\tau})]\!]$$

Proof by corollary 5 and 7.

With the help of the above results we can show that the folding rule $v \in \mathrm{Val}^{\sigma(\mathrm{Rec}\, X.\sigma(X))} \to v \in \mathrm{Val}^{\mathrm{Rec}\, X.\sigma(X)}$ fo syntactic values is also valid for semantic values. Furthermore the opposite direction of this rule is valid as well.

Corollary 9 (Folding rule for semantic values)

$$\mathrm{fold}\,([\![\sigma(\mathrm{Rec}\, X.\sigma(X))]\!]) = [\![\mathrm{Rec}\, X.\sigma(X)]\!]$$

Proof. By corollary 7 $[\![\sigma(\mathrm{Rec}\, X.\sigma(X))]\!] = [\![\sigma(X)]\!]_{[\![\mathrm{Rec}\, X.\sigma(X)]\!]}$ and $[\![\mathrm{Rec}\, X.\sigma(X)]\!]$ is the least fixed point of fold $\circ\, [\![\sigma(X)]\!]_{\_}$.

Codomain and termination. We define the *codomain* of a function $f \in [\![\sigma \to \tau(\vec{X})]\!]_{\vec{V}}$ as

$$\mathrm{CoDom}(f) := \left\{v \in [\![\tau(\vec{X})]\!]_{\vec{V}} : \exists u \in [\![\sigma]\!].\, f@u \Downarrow v\right\}$$

We say a closure $c \in \mathrm{Cl}^\sigma$ *terminates* iff it evaluates to some value $v \in [\![\sigma]\!]$:

$$c \Downarrow :\leftrightarrow \exists v \in [\![\sigma]\!].\, c \Downarrow v$$

4.1 Example

To clarify the semantics of a recursive type, we will consider $[\![\mathbf{Nat}]\!]$ and show that it contains all numerals $\overline{n} \in \mathrm{Val}^{\mathbf{Nat}}$:

$$[\![\mathbf{Nat}]\!] = \mathrm{Val}^{\mathbf{Nat}}$$

Since $\mathbf{Nat} \equiv \mathrm{Rec}\, X.1 + X$ we have to show that $\mathrm{Val}^{\mathbf{Nat}}$ is least fixed point of

$$\begin{array}{rcl}
\mathcal{F} & : & \mathcal{P}\left(\mathrm{Val}^{\mathbf{Nat}}\right) \to \mathcal{P}\left(\mathrm{Val}^{\mathbf{Nat}}\right) \\
\mathcal{F}(W) & := & \{\mathrm{fold}(v) : v \in [\![1 + X]\!]_W\} \\
 & = & \{\mathrm{fold}(\mathrm{inl}()), \mathrm{fold}(\mathrm{inr}(v)) : v \in W\} \\
 & = & \{\mathbf{O}, \mathbf{S}(v) : v \in W\}
\end{array}$$

Since trivially $\mathcal{F}(\mathrm{Val}^{\mathbf{Nat}}) \subseteq \mathrm{Val}^{\mathbf{Nat}}$, $\mathrm{Val}^{\mathbf{Nat}}$ is pre-fixed point of $\mathcal{F}$. We only have to show that there is no smaller pre-fixed point W. To this end, we show for all pre-fixed points $W \subseteq \mathrm{Val}^{\mathbf{Nat}}$

$$\begin{array}{ll}
\text{(i)} & \bigcup_{n \in \mathbb{N}} \mathcal{F}^n(\emptyset) \subseteq W \\
\text{(ii)} & \mathrm{Val}^{\mathbf{Nat}} \subseteq \bigcup_{n \in \mathbb{N}} \mathcal{F}^n(\emptyset)
\end{array}$$

Proof. (i) We show $\forall n \in \mathbb{N}.\ \mathcal{F}^n(\emptyset) \subseteq W$ by induction on n:

$n = 0$: $\mathcal{F}^0(\emptyset) = \emptyset \subseteq W$

$n \to n+1$: By induction hypothesis $\mathcal{F}^n(\emptyset) \subseteq W$, hence by monotonicity of $\mathcal{F}$ and (ispfp) for W we obtain $\mathcal{F}^{n+1}(\emptyset) = \mathcal{F}(\mathcal{F}^n(\emptyset)) \subseteq \mathcal{F}(W) \subseteq W$.

Hence it follows that $\bigcup_{n \in \mathbb{N}} \mathcal{F}^n(\emptyset) \subseteq W$. (ii) We show $\forall n \in \mathbb{N}.\ \mathcal{F}^n(\emptyset) = \{\overline{m} : m < n\}$ by induction on n:

$n = 0$: $\mathcal{F}^0(\emptyset) = \emptyset$

$n \to n{+}1$: By induction hypothesis $\mathcal{F}^n(\emptyset) = \{\overline{m} : m < n\}$, thus $\mathcal{F}^{n+1}(\emptyset) = \mathcal{F}(\{\overline{m} : m < n\}) = \{\mathbf{O}, \mathbf{S}(\overline{m}) : m < n\} = \{\overline{m} : m < n+1\}$.

Hence it follows that $\mathrm{Val}^{\mathbf{Nat}} = \bigcup_{n\in\mathbb{N}} \{\overline{m} : m < n\} \subseteq \bigcup_{n\in\mathbb{N}} \mathcal{F}^n(\emptyset)$. □

5 Wellfoundedness of the structural ordering

We now define a transitive structural ordering $<$ on the semantic values. The basic idea is that a value v is structurally smaller than a value w if the representing tree of v is a subtree of w.

In our approach the order of a value is only decreased ($<$) by case analysis, whereas projection and application keep it on the same level ($\leq$). This has to be explained: Since we want to show that a function $f \in \mathrm{Val}^{\sigma\to\tau}$ terminates on an input $v \in [\![\sigma]\!]$ if it terminates on all $w \in [\![\sigma]\!]$ that are structurally smaller than v, i.e. $w < v$, we only want to be able to compare values of the same type σ. Since injection (inl, inr), pairing and building functions by λ enlarge the type and only folding shrinks the type, we need at least one folding step to obtain a greater value v of the same type σ out of a given value w. Therefore σ is a recursive type and has at least one "Rec" in its component. But since only recursive types over *sums* are both non-trivial and non-empty, these are the only types of interest for our work. Thus we can conclude that on the way of generating v out of w there is at least one injection, which then actually increases ($<$) the order. (See the example at the end of this section.)

Structural ordering. We define a pair of mutually dependent families of relations $<_{\sigma,\tau}, \leq_{\sigma,\tau} \subseteq [\![\sigma]\!] \times [\![\tau]\!]$ inductively as follows:

(lerefl) $$\frac{}{v \leq_{\sigma,\sigma} v}$$

(lelt) $$\frac{w <_{\sigma,\tau} v}{w \leq_{\sigma,\tau} v}$$

(ltinl) $$\frac{w \leq_{\rho,\sigma} v}{w <_{\rho,\sigma+\tau} \mathrm{inl}(v)}$$

(ltinr) $$\frac{w \leq_{\rho,\tau} v}{w <_{\rho,\sigma+\tau} \mathrm{inr}(v)}$$

(ltfst) $$\frac{w <_{\rho,\sigma} v}{w <_{\rho,\sigma\times\tau} (v, v')}$$

(ltsnd) $$\frac{w <_{\rho,\tau} v'}{w <_{\rho,\sigma\times\tau} (v, v')}$$

(lefst) $$\frac{w \leq_{\rho,\sigma} v}{w \leq_{\rho,\sigma\times\tau} (v, v')}$$

(lesnd) $$\frac{w \leq_{\rho,\tau} v'}{w \leq_{\rho,\sigma\times\tau} (v, v')}$$

(ltarr) $$\frac{\exists v \in \mathrm{CoDom}(f). w <_{\rho,\tau} v}{w <_{\rho,\sigma\to\tau} f}$$

(learr) $$\frac{\exists v \in \mathrm{CoDom}(f). w \leq_{\rho,\tau} v}{w \leq_{\rho,\sigma\to\tau} f}$$

$$\text{(ltfold)}\quad \frac{w <_{\sigma,\tau(\mathrm{Rec}\,X.\tau(X))} v}{w <_{\sigma,\mathrm{Rec}\,X.\tau(X)} \mathrm{fold}(v)} \qquad \text{(lefold)}\quad \frac{w \leq_{\sigma,\tau(\mathrm{Rec}\,X.\tau(X))} v}{w \leq_{\sigma,\mathrm{Rec}\,X.\tau(X)} \mathrm{fold}(v)}$$

By definition $<$ and $\leq$ are transitive, $<$ is irreflexive and $\leq$ reflexive.

Notation. Since the indexes σ and τ of $<_{\sigma,\tau}$ and $\leq_{\sigma,\tau}$ are determined in most expressions, we omit them for better readability.

We want to show that every set of semantic values is wellfounded which is given if every value is accessible. Therefore we first define the sets of accessible values w.r.t. $<$ and then we show that each value is in the accessible set of its type.

Accessible sets. We define the family of accessible sets $\mathrm{Acc}^\sigma \subseteq [\![\sigma]\!]$ w.r.t. $<$ inductively as follows. (This is the usual definition.)

$$\text{(acc)}\qquad \frac{\forall\tau, [\![\tau]\!] \ni w < v.\, w \in \mathrm{Acc}^\tau}{v \in \mathrm{Acc}^\tau}$$

Notation. For lists $\vec{\rho}$ of types we use the abbreviation $\mathrm{Acc}^{\vec{\rho}} := \mathrm{Acc}^{\rho_1}, \ldots, \mathrm{Acc}^{\rho_n}$.

Proposition 10 (Destructors for Acc) *If a value is accessible, then smaller values are accessible as well:*

$$\begin{array}{rl}
(\mathrm{acc}^{-1}) & \forall v \in \mathrm{Acc}^\sigma, [\![\tau]\!] \ni w < v.w \in \mathrm{Acc}^\tau \\
(\mathrm{accfst}) & \forall (v, v') \in \mathrm{Acc}^{\sigma\times\tau}.v \in \mathrm{Acc}^\sigma \\
(\mathrm{accsnd}) & \forall (v, v') \in \mathrm{Acc}^{\sigma\times\tau}.v' \in \mathrm{Acc}^\tau \\
(\mathrm{accres}) & \forall f \in \mathrm{Acc}^{\sigma\to\tau}.\mathrm{CoDom}(f) \subseteq \mathrm{Acc}^\tau \\
(\mathrm{accunf}) & \forall\, \mathrm{fold}(v) \in \mathrm{Acc}^{\mathrm{Rec}\,X.\sigma(X)}.v \in \mathrm{Acc}^{\sigma(\mathrm{Rec}\,X.\sigma(X))}
\end{array}$$

Proof.

(acc^{-1}) We assume $v \in \mathrm{Acc}^\sigma$. Since there is only one constructor (acc) for accessible sets, we can use it in the reverse direction and get $[\![\tau]\!] \ni w < v.w \in \mathrm{Acc}^\tau$, what we had to show.

(accfst) To prove accessibility of v we have to show $\forall [\![\rho]\!] \ni w < v.w \in \mathrm{Acc}^\rho$. Be now $[\![\rho]\!] \ni w < v$. By (ltfst) we get $w < (v, v')$, and since (v, v') is accessible by assumption, $w \in \mathrm{Acc}^\rho$ by (acc^{-1}).

(accsnd) analogously

(accres) We have to show $v \in \mathrm{CoDom}(f) \to v \in \mathrm{Acc}^\tau$. Assume $[\![\rho]\!] \ni w < v \in \mathrm{CoDom}(f)$. By (ltarr) we get $w < f$ and again (acc^{-1}) proves $w \in \mathrm{Acc}^\rho$, hence $v \in \mathrm{Acc}^\tau$.

(accunf) Again we have to show $\forall [\![\rho]\!] \ni w < v. w \in \mathrm{Acc}^\rho$. Be now $w < v$. By (ltfold) we get $w < \mathrm{fold}(v)$, and since $\mathrm{fold}(v)$ is accessible by assumption, $w \in \mathrm{Acc}^\rho$ by (acc^{-1}). □

Proposition 11 (Accessibility of less-equal values) *The values less than or equal to an accessible value are accessible themselves.*

$$\text{(accle)} \qquad \forall v \in \mathrm{Acc}^\tau, [\![\sigma]\!] \ni w \leq v. w \in \mathrm{Acc}^\sigma$$

Proof by induction on generation of $w \leq v$:

(lerefl) Pattern matches with $\sigma \equiv \tau$, $w \equiv v$: $v \leq v \in \mathrm{Acc}^\tau$.

(lelt) $w < v$: We get $w \in \mathrm{Acc}^\sigma$ by (acc^{-1}).

(lefst) Pattern matches with $\tau \equiv \sigma' \times \tau'$, $v \equiv (v', w')$: We get $v' \in \mathrm{Acc}^{\sigma'}$ by (accfst) and $w \in \mathrm{Acc}^\sigma$ by the induction hypothesis $w \leq v' \to w \in \mathrm{Acc}^\sigma$.

(lesnd) analogously

(learr) Pattern matches with $\tau \equiv \sigma' \to \tau'$: There is a $v' \in \mathrm{CoDom}(v)$ with $w \leq v'$. Since by (accres) $v' \in \mathrm{Acc}^{\tau'}$, $w \in \mathrm{Acc}^\sigma$ by induction hypothesis.

(lefold) Pattern matches with $\tau \equiv \mathrm{Rec}\, X.\sigma'(X)$, $v \equiv \mathrm{fold}(v')$. Now $v' \in \mathrm{Acc}^{\sigma(\mathrm{Rec}\, X.\sigma'(X))}$ by (accunf) and thus $w \in \mathrm{Acc}^\sigma$ by induction hypothesis. □

Proposition 12 (Accessibility of successors) *The successors of accessible values are accessible as well.*

$$\begin{array}{rl}
\text{(accinl)} & \forall v \in \mathrm{Acc}^\sigma.\, \mathrm{inl}(v) \in \mathrm{Acc}^{\sigma+\tau} \\
\text{(accinr)} & \forall v \in \mathrm{Acc}^\tau.\, \mathrm{inr}(v) \in \mathrm{Acc}^{\sigma+\tau} \\
\text{(accpair)} & \forall v \in \mathrm{Acc}^\sigma, v' \in \mathrm{Acc}^\tau.\, (v, v') \in \mathrm{Acc}^{\sigma\times\tau} \\
\text{(accarr)} & \forall f \in [\![\sigma \to \tau]\!].\, (\mathrm{CoDom}(f) \subseteq \mathrm{Acc}^\tau) \to f \in \mathrm{Acc}^{\sigma\to\tau} \\
\text{(accfold)} & \mathrm{fold}\left(\mathrm{Acc}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}\right) \subseteq \mathrm{Acc}^{\mathrm{Rec}\, X.\sigma(X)}
\end{array}$$

Proof. We show all propositions by induction over the structural ordering.

(accinl) To prove $\mathrm{inl}(v) \in \mathrm{Acc}^{\sigma+\tau}$ we have to show $\forall [\![\rho]\!] \ni w < \mathrm{inl}(v).w \in \mathrm{Acc}^{\rho}$ by induction over the generation of $w < \mathrm{inl}(v)$. Only case (ltinl) matches: $w \leq v$. By (accle) we get $w \in \mathrm{Acc}^{\rho}$.

(accinr) analogously

(accpair) We show $\forall [\![\rho]\!] \ni w < (v, v').w \in \mathrm{Acc}^{\rho}$ by induction over the generation of $w < (v, v')$. The first matching case (ltfst) gives us $w < v$, and since $v \in \mathrm{Acc}^{\sigma}$ also $w \in \mathrm{Acc}^{\rho}$. Case (ltsnd) analogous.

(accarr) Induction over $[\![\rho]\!] \ni w < f$ gives us by (ltarr) a $v \in \mathrm{CoDom}(f)$ with $w < v$. Since by assumption $v \in \mathrm{Acc}^{\tau}$, we have $w \in \mathrm{Acc}^{\rho}$. Hence $f \in \mathrm{Acc}^{\sigma\to\tau}$.

(accfold) We must show $\forall v \in \mathrm{Acc}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}.\mathrm{fold}(v) \in \mathrm{Acc}^{\mathrm{Rec}\, X.\sigma(X)}$. Given an arbitrary value w, $w < \mathrm{fold}(v)$ is generated out of $w < v$ by (ltfold). Again by assumption $v \in \mathrm{Acc}^{\sigma(\mathrm{Rec}\, X.\sigma(X))}$, thus w is accessible and hence $\mathrm{fold}(v) \in \mathrm{Acc}^{\mathrm{Rec}\, X.\sigma(X)}$. □

Now we can show that all semantic values are accessible. It would be sufficient to know that all semantic values of closes types $[\![\sigma]\!]$ are accessible. But to prove it for recursive types we have to show the stronger proposition that also the semantic values of open types are accessible, where we insert sets of accessible values for the free type variables.

Lemma 13 (All semantic values are accessible)

$$\forall \sigma(\vec{X}), \vec{\rho}.\, [\![\sigma(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}} \subseteq \mathrm{Acc}^{\sigma(\vec{\rho})}$$

Proof by induction over $\sigma(\vec{X})$.

(Unit) $[\![1]\!] = \{()\} \subseteq \mathrm{Acc}^1$. We have to show $\forall [\![\rho]\!] \ni w < ().w \in \mathrm{Acc}$. This is trivially true because there is no matching rule for $w < ()$.

(Var) $[\![X_n]\!]_{\mathrm{Acc}^{\vec{\rho}}} = \mathrm{Acc}^{\rho_n} = \mathrm{Acc}^{X_n[\vec{X}:=\vec{\rho}]}$

(Weak) $[\![\sigma(\vec{X}, X)]\!]_{\mathrm{Acc}^{\vec{\rho},\rho}} = [\![\sigma(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}} \subseteq \mathrm{Acc}^{\sigma[\vec{X}:=\vec{\rho}]} = \mathrm{Acc}^{\sigma[\vec{X}:=\vec{\rho},X:=\rho]}$ by IH.

(Sum) Be $v \in [\![(\sigma + \tau)(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}}$. Case $v = \mathrm{inl}(v'), v' \in [\![\sigma(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}}$: By IH we have $v' \in \mathrm{Acc}^{\sigma(\vec{\rho})}$, hence by (accinl) $v' \in \mathrm{Acc}^{(\sigma+\tau)(\vec{\rho})}$. Case $v = \mathrm{inr}(v)$ analogous.

(Prod) Be $(v, v') \in [\![(\sigma \times \tau)(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}}$. Since $v \in [\![\sigma(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}})}$ and $v' \in [\![\tau(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}})}$, by IH $v \in \mathrm{Acc}^{\sigma(\vec{\rho})}$ and $v' \in \mathrm{Acc}^{\tau(\vec{\rho})}$, hence by (accpair) $(v, v') \in \mathrm{Acc}^{(\sigma\times\tau)(\vec{\rho})}$.

(Arr) Be $f \in [\![\sigma \to \tau(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}}$. By IH we have $\mathrm{CoDom}(f) \subseteq [\![\tau(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}} \subseteq \mathrm{Acc}^{\tau(\vec{\rho})}$. By Corollary 8 we get $f \in [\![\sigma \to \tau(\vec{\rho})]\!]$ and hence by (accarr) $f \in \mathrm{Acc}^{\sigma\to\tau(\vec{\rho})}$.

(Rec) We define $\mathcal{F}(W) := \mathrm{fold}\left([\![\sigma(\vec{X}, Y)]\!]_{\mathrm{Acc}^{\vec{\rho}}, W}\right)$, hence our goal is $[\![\mathrm{Rec}\,\sigma(\vec{X}, Y)]\!]_{\mathrm{Acc}^{\vec{\rho}}} = \mathrm{lfp}\,\mathcal{F} \subseteq \mathrm{Acc}^{\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y)}$. We instantiate the induction hypothesis $[\![\sigma(\vec{X}, Y)]\!]_{\mathrm{Acc}^{\vec{\rho}},\mathrm{Acc}^{\rho}} \subseteq \mathrm{Acc}^{\sigma(\vec{\rho},\rho)}$ for $\rho \equiv \mathrm{Rec}\,Y.\sigma(\vec{\rho}, Y)$ and get by monotonicity of fold $\mathcal{F}(\mathrm{Acc}^{\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y)}) \subseteq \mathrm{fold}\left(\mathrm{Acc}^{\sigma(\vec{\rho},\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y))}\right)$. Applying (accfold) yields $\mathcal{F}(\mathrm{Acc}^{\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y)}) \subseteq \mathrm{Acc}^{\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y)}$, hence $\mathrm{Acc}^{\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y)}$ is pre-fixed point of $\mathcal{F}$. By (ismpfp) we know that $\mathrm{lfp}\,\mathcal{F} \subseteq \mathrm{Acc}^{\mathrm{Rec}\,Y.\sigma(\vec{\rho},Y)}$. □

Corollary 14

$$\forall \sigma(\vec{X}), \vec{\rho}.\, [\![\sigma(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}} = \mathrm{Acc}^{\sigma(\vec{\rho})}$$

Proof. "$\subseteq$" by lemma, "$\supseteq$": Since for closed types ρ from the lemma we immediately get $[\![\rho]\!] = \mathrm{Acc}^{\rho}$, we have by corollary 7 $\mathrm{Acc}^{\sigma(\vec{\rho})} = [\![\sigma(\vec{\rho})]\!] = [\![\sigma(\vec{X})]\!]_{[\![\vec{\rho}]\!]} = [\![\sigma(\vec{X})]\!]_{\mathrm{Acc}^{\vec{\rho}}}$.

5.1 Example

As an application of the results of this section we want to prove that the addition on ordinal numbers **addOrd** $\in \mathrm{Tm}^{\mathbf{Ord}\to\mathbf{Ord}\to\mathbf{Ord}}$ defined in section 2.3.2 is structurally recursive. Therefore we must show that for each (semantic) input value recursive calls take place only with structurally smaller (semantic) values. In the following we will identify **addOrd** with its value $\langle \mathbf{addOrd}; \cdot\rangle \in \mathrm{Val}^{\mathbf{Ord}\to\mathbf{Ord}\to\mathbf{Ord}}$.
There are three possible cases for the input $v \in [\![\mathbf{Ord}]\!]$:

$v \equiv \mathbf{O}$: No recursive call.

$v \equiv \mathbf{S}(v')$: $v' \in [\![\mathbf{Ord}]\!]$. We have one recursive call with argument v', which is structurally smaller than v:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v' \leq v'}\,\text{lerefl}}{v' < \mathrm{inr}(v')}\,\text{ltinr}}{v' \leq \mathrm{inr}(v')}\,\text{lelt}}{v' < \mathrm{inl}(\mathrm{inr}(v'))}\,\text{ltinl}}{v' < \mathbf{S}(v') \equiv \mathrm{fold}(\mathrm{inl}(\mathrm{inr}(v')))}\,\text{ltfold}$$

$v \equiv \mathbf{Lim}(f)$: $f \in [\![\mathbf{Nat} \to \mathbf{Ord}]\!]$. Here again we have one recursive call. The argument is $(f\ z)$ which for $z \in [\![\mathbf{Nat}]\!]$ evaluates to a $w \in \mathrm{CoDom}(f)$. Thus we can derive $w < \mathbf{Lim}(f)$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{w \leq w}\,\text{lerefl}}{\exists v' \in \mathrm{CoDom}(f).\, w \leq v'}\,\exists\mathcal{I}}{w \leq f}\,\text{learr}}{w < \mathrm{inr}(f)}\,\text{ltinr}}{w < \mathbf{Lim}(f) \equiv \mathrm{fold}(\mathrm{inr}(f))}\,\text{ltfold}$$

$\square$

6 Soundness of the termination criterium

We want to define a subset TM of good terms and prove normalization for all $t \in \mathrm{TM}$. Since the recursive terms are candidates of nontermination, we have to restrict ourselves to structurally recursive terms for that we will prove termination easily by the wellfoundedness of the structural ordering. The normalization proof then can be done mechanically.

Structurally recursive terms. The subset of *environments of semantic values* over context $\Gamma = x_1^{\sigma_1}, \dots, x_n^{\sigma_n}$ is of course

$$[\![\Gamma]\!] = \{x_1 = v_1, \dots, x_n = v_n : v_i \in [\![\sigma_i]\!]\} \subseteq \mathrm{Val}(\Gamma)$$

We will refer to an $e \in [\![\Gamma]\!]$ as a *good environment*. We define the set of structurally recursive terms $\mathrm{SR}^{\sigma\to\tau}[\Gamma]$ of type $\sigma \to \tau$ over context Γ as the recursive terms that applied to any value v terminate in any good environment under the condition that they terminate for all structurally smaller values $w < v$:

$$\begin{array}{rcl} \mathrm{SR}^{\sigma\to\tau}[\Gamma] & := & \{\mathrm{rec}\, g.t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma] : \forall e \in [\![\Gamma]\!], v \in [\![\sigma]\!]. \\ & & (\forall [\![\sigma]\!] \ni w < v. \langle \mathrm{rec}\, g.t; e\rangle @ w \Downarrow) \to \langle \mathrm{rec}\, g.t; e\rangle @ v \Downarrow\} \end{array}$$

We now want to prove that all structurally recursive terms supplied with any good environment are semantic values, i.e. terminate for every semantic value we apply. Since we have proved wellfoundedness of $<$, this can easily be done by

Wellfounded induction for families of predicates. We present a formulation of the wellfounded-part-induction principle for our definition of Acc^σ as family of accessible sets indexed by type σ. Let P^σ be a family of predicates over values of type σ.

$$(\mathrm{accind}') \qquad \frac{\forall v \in [\![\sigma]\!]. (\forall \rho, [\![\rho]\!] \ni w < v. P^\rho(w)) \to P^\sigma(v)}{\forall v \in \mathrm{Acc}^\sigma. P^\sigma(v)}$$

For our purpose a specialization of this induction principle for a single predicate over a fixed type is sufficient. Given a type σ and a predicate $P \subseteq [\![\sigma]\!]$ we obtain

$$(\mathrm{accind}) \qquad \frac{\forall v \in [\![\sigma]\!]. (\forall [\![\sigma]\!] \ni w < v. P(w)) \to P(v)}{\forall v \in \mathrm{Acc}^\sigma. P(v)}$$

by defining a family of predicates $P^\rho \subseteq [\![\rho]\!]$ as

$$P^\rho := \begin{cases} P & \text{if } \rho = \sigma \\ [\![\rho]\!] & \text{else} \end{cases}$$

and applying (accind′). We see that the conditions $\forall \rho, [\![\rho]\!] \ni w < v.\, P^\rho(w)$ and $\forall [\![\sigma]\!] \ni w < v.\, P(w)$ are equivalent by definition of P^ρ.

Lemma 15 (Structurally recursive functions terminate)

$$\forall \mathrm{rec}\, g.t \in \mathrm{SR}^{\sigma\to\tau}, e \in [\![\Gamma]\!].\, \langle \mathrm{rec}\, g.t; e\rangle \in [\![\sigma \to \tau]\!]$$

Proof. Our goal is by definition equivalent to

$$\forall \mathrm{rec}\, g.t \in \mathrm{SR}^{\sigma\to\tau}, e \in [\![\Gamma]\!], v \in [\![\sigma]\!].\, \langle \mathrm{rec}\, g.t; e\rangle @ v \Downarrow$$

For a fixed environment e the assumption $\mathrm{rec}\, g.t \in \mathrm{SR}^{\sigma\to\tau}$ expands to

$$\forall v \in [\![\sigma]\!].\, (\forall [\![\sigma]\!] \ni w < v.\, \langle \mathrm{rec}\, g.t; e\rangle @ w \Downarrow) \to \langle \mathrm{rec}\, g.t; e\rangle @ v \Downarrow$$

By (accind) with $P(v) \equiv \langle \mathrm{rec}\, g.t; e\rangle @ v \Downarrow$ we get

$$\forall v \in \mathrm{Acc}^\sigma.\, \langle \mathrm{rec}\, g.t; e\rangle @ v \Downarrow$$

which is equivalent to our claim because of $\mathrm{Acc}^\sigma = [\![\sigma]\!]$.

Good terms. We inductively define the set of good terms $\mathrm{TM}^\sigma[\Gamma] \subset \mathrm{Tm}^\sigma[\Gamma]$ of type σ over context Γ, i.e. the terms that ensure termination. These rules are almost identical to the original term formation rules (see page 12), we only change Tm to TM and label the rule in CAPITAL letters, e.g.

$$(\text{PAIR}) \qquad \frac{s \in \mathrm{TM}^\sigma[\Gamma] \quad t \in \mathrm{TM}^\tau[\Gamma]}{(s,t) \in \mathrm{TM}^{\sigma\times\tau}[\Gamma]}$$

Only the rule (rec) is replaced by

$$(\text{REC}) \qquad \frac{t \in \mathrm{TM}^{\sigma\to\tau}[\Gamma, g^{\sigma\to\tau}] \quad \mathrm{rec}\, g.t \in \mathrm{SR}^{\sigma\to\tau}[\Gamma]}{\mathrm{rec}\, g.t \in \mathrm{TM}^{\sigma\to\tau}[\Gamma]}$$

Note that in the second condition $\mathrm{rec}\, g.t \in \mathrm{SR}^{\sigma\to\tau}[\Gamma]$ we use the natural embedding $\mathrm{TM}^\sigma[\Gamma] \hookrightarrow \mathrm{Tm}^\sigma[\Gamma]$ that can be defined by recursion over TM simultanously to the inductive definition in the obvious way.

Good closures. Consequently the set of good closures CL^τ of type τ is defined as

$$\begin{array}{rcl}\mathrm{Cl}^\tau & := & \{\langle t; e\rangle : \Gamma \in \mathrm{Cxt}, t \in \mathrm{TM}^\tau[\Gamma], e \in [\![\Gamma]\!]\} \\ & \cup & \{f@u : f \in [\![\sigma \to \tau]\!], u \in [\![\sigma]\!]\}\end{array}$$

Again $\mathrm{CL}^\tau \subseteq \mathrm{Cl}^\tau$, hence we can use our operational semantics $\Downarrow$ on good closures as well. Now no obstacle is in our way to show

Theorem 16 (Normalization)

$$\forall \sigma, \Gamma, t \in \mathrm{TM}^\sigma[\Gamma], e \in [\![\Gamma]\!]. \langle t; e\rangle \Downarrow$$

Proof by induction on $t \in \mathrm{TM}^\sigma[\Gamma]$. We overload the definition of e^Γ and now mean $e \in [\![\Gamma]\!]$.

(UNIT) By (opunit) $\langle (); \cdot\rangle \Downarrow () \in [\![1]\!]$.

(VAR) Since $e^\Gamma, x^\sigma = v \in [\![\Gamma, x^\sigma]\!]$ by (opvar) we prove $\langle x[\Gamma, x^\sigma]; e, x = v\rangle \Downarrow v \in [\![\sigma]\!]$.

(WEAK) By IH there is a $v \in [\![\sigma]\!]$ such that $\langle t^\sigma[\Gamma]; e\rangle \Downarrow v$, hence by (opweak) $\langle t[\Gamma, x]; e, x = w\rangle \Downarrow v \in [\![\sigma]\!]$.

(INL) By IH $\langle t^\sigma; e\rangle \Downarrow v \in [\![\sigma]\!]$, thus by (opinl) $\langle \mathrm{inl}(t)^{\sigma+\tau}; e\rangle \Downarrow \mathrm{inl}(v) \in [\![\sigma + \tau]\!]$.

(INR) analogously

(CASE) We must show $\langle \mathrm{case}(t^{\sigma+\tau}[\Gamma], x^\sigma.(l^\rho[\Gamma, x^\sigma]), y^\tau.(r^\rho[\Gamma, y^\tau])); e\rangle \Downarrow v \in [\![\rho]\!]$. By IH we have $\langle t^{\sigma+\tau}[\Gamma]; e\rangle \Downarrow w' \in [\![\sigma + \tau]\!]$. Case $w' = \mathrm{inl}(w^\sigma)$: By IH we get $\langle l^\rho[\Gamma, x^\sigma]; e, x = w\rangle \Downarrow v \in [\![\rho]\!]$, hence by (opcasel) we prove our claim. Case $w' = \mathrm{inr}(w^\tau)$ analogously by (opcaser).

(PAIR) Here we show $\langle (s^\sigma, t^\tau); e\rangle \Downarrow$. By IH we have $\langle s; e\rangle \Downarrow v \in [\![\sigma]\!]$ and $\langle t; e\rangle \Downarrow w \in [\![\tau]\!]$ hence by (oppair) $\langle (s, t); e\rangle \Downarrow (v, w)$ which is in $[\![\sigma \times \tau]\!]$ by definition.

(FST) By IH $\langle p^{\sigma\times\tau}; e\rangle \Downarrow (v, w) \in [\![\sigma \times \tau]\!]$, hence by (opfst) $\langle \mathrm{fst}(p); e\rangle \Downarrow v \in [\![\sigma]\!]$.

(SND) analogously

(LAM) By (oplam) $\langle \lambda x^\sigma.t^\tau; e^\Gamma\rangle \Downarrow \langle \lambda x.t; e\rangle$. We have to show $\langle \lambda x.t; e\rangle \in [\![\sigma \to \tau]\!]$, i.e. $\langle \lambda x.t; e\rangle @ u \Downarrow$ for all $u \in [\![\sigma]\!]$. Refinement by (opappvl) reduces our goal to $\langle t; e, x = u\rangle \Downarrow$ which we get by the induction hypothesis.

(REC) Here by (oprec) we have to show $\langle \mathrm{rec}\, g.t; e\rangle \in [\![\sigma \to \tau]\!]$, which is true by lemma 15 since $\mathrm{rec}\, g.t \in \mathrm{SR}^{\sigma\to\tau}$ by definition.

(APP) By IH we have $\langle t^{\sigma\to\tau}; e\rangle \Downarrow f \in [\![\sigma \to \tau]\!]$ and $\langle s; e\rangle \Downarrow u \in [\![\sigma]\!]$ therefore $f @ u \Downarrow$ and by (opapp) $\langle t\, s; e\rangle \Downarrow$ as well.

(FOLD) By IH $\langle t; e\rangle \Downarrow v \in [\![\sigma(\mathrm{Rec}\, X.\sigma(X))]\!]$, hence by (opfold) $\langle \mathrm{fold}(t); e\rangle \Downarrow \mathrm{fold}(v)$, which is in $[\![\mathrm{Rec}\, X.\sigma(X)]\!]$ by the folding rule for semantic values (corollary 9).

(UNFOLD) By IH $\langle t; e\rangle \Downarrow \mathrm{fold}(v) \in [\![\mathrm{Rec}\, X.\sigma(X)]\!]$, hence by (opunfold) $\langle \mathrm{unfold}(t); e\rangle \Downarrow v$, which is in $[\![\sigma(\mathrm{Rec}\, X.\sigma(X))]\!]$ by corollary 9.

7 Extensions

In the following we want to extend our soundness proof to positive types, functions with multiple parameters and mutual recursion in order to cover the full functionality of the `foetus` termination checker.

7.1 Positive types

In the main part of our work we have restricted ourselves to a system with only strictly positive types. But we can without much effort expand the proof of wellfoundedness and therefore the termination criterium to systems with positive inductive types. In the following we will give the modifications to definitions and lemmata such that positive types can be handled as well.

Types. We now distinct between these type variables that appear positive and those that appear negative in types. Because again we do not support polymorphic types and thus need free type variables only for definition of positive recursive types, a type variable may appear either positive or negative (unlike the common definition in System F, where variables may be seen as both positive and negative or neither positive nor negative in some cases).

So we define the family of sets of types $\mathrm{Ty}(\vec{X};\vec{Y})$ over the finite sets of positively appearing variables $\vec{X}$ and negatively appearing variables $\vec{Y}$ inductively as follows:

$$\text{(Unit)}\quad \frac{}{1 \in \mathrm{Ty}(\emptyset;\emptyset)} \qquad \text{(Var)}^{9}\quad \frac{\vec{X}, X \subset \mathrm{TyVars}}{X \in \mathrm{Ty}(\vec{X}, X;\emptyset)}$$

$$\text{(Weak}^{+}\text{)}\quad \frac{\sigma \in \mathrm{Ty}(\vec{X};\vec{Y}) \quad X \notin \vec{X},\vec{Y}}{\sigma \in \mathrm{Ty}(\vec{X}, X;\vec{Y})}$$

$$\text{(Weak}^{-}\text{)}\quad \frac{\sigma \in \mathrm{Ty}(\vec{X};\vec{Y}) \quad Y \notin \vec{X},\vec{Y}}{\sigma \in \mathrm{Ty}(\vec{X};\vec{Y}, Y)}$$

$$\text{(Sum)}\quad \frac{\sigma,\tau \in \mathrm{Ty}(\vec{X};\vec{Y})}{\sigma + \tau \in \mathrm{Ty}(\vec{X};\vec{Y})} \qquad \text{(Prod)}\quad \frac{\sigma,\tau \in \mathrm{Ty}(\vec{X};\vec{Y})}{\sigma \times \tau \in \mathrm{Ty}(\vec{X};\vec{Y})}$$

[9] Note that there is no rule like (Var^{-}) since in the type $\sigma \equiv X$ created by the rule (Var) the variable X appears positively.

$$(\mathrm{Arr})^{10} \quad \frac{\sigma \in \mathrm{Ty}(\vec{Y};\vec{X}) \quad \tau \in \mathrm{Ty}(\vec{X};\vec{Y})}{\sigma \to \tau \in \mathrm{Ty}(\vec{X};\vec{Y})}$$

$$(\mathrm{Rec}) \quad \frac{\sigma \in \mathrm{Ty}(\vec{X},X;\vec{Y})}{\mathrm{Rec}X.\sigma \in \mathrm{Ty}(\vec{X};\vec{Y})}$$

Substitution. We now have to define substitution of the positive $\vec{X}$ and negative $\vec{Y}$ type variables in a type $\sigma(\vec{X},\vec{Y})$ by lists of types $\vec{\xi}$ and $\vec{\eta}$ over the positive $\vec{X'}$ and negative $\vec{Y'}$ variables. For $\sigma[\vec{X} := \vec{\xi}; \vec{Y} := \vec{\eta}]$ to be in $\mathrm{Ty}(\vec{X'};\vec{Y'})$, it is necessary that $\vec{\xi} \subset \mathrm{Ty}(\vec{X'};\vec{Y'})$ and $\vec{\xi} \subset \mathrm{Ty}(\vec{Y'};\vec{X'})$. The complete modified definition:

(Unit) $1[] := 1$

(Var) $X[\vec{X} := \vec{\xi}, X := \xi;] := \xi$

(Weak$^+$) $\sigma(\vec{X},X;\vec{Y})[\vec{X},X := \vec{\rho},\rho;\vec{Y} := \vec{\eta}] := \sigma(\vec{X})[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}]$

(Weak$^-$) $\sigma(\vec{X};\vec{Y},Y)[\vec{X} := \vec{\rho};\vec{Y},Y := \vec{\eta},\eta] := \sigma(\vec{X})[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}]$

(Sum) $\left(\sigma(\vec{X};\vec{Y}) + \tau(\vec{X};\vec{Y})\right)[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}] :=$
$\sigma[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}] + \tau[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}]$

(Prod) $\left(\sigma(\vec{X};\vec{Y}) \times \tau(\vec{X};\vec{Y})\right)[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}] :=$
$\sigma[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}] \times \tau[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}]$

(Arr) $\left(\sigma(\vec{Y};\vec{X}) \to \tau(\vec{X};\vec{Y})\right)[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}] :=$
$\sigma[\vec{Y} := \vec{\eta};\vec{X} := \vec{\xi}] \to \tau[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}]$

(Rec) We can assume $Z \notin \vec{X'},\vec{Y'}$ by the renaming convention.
$\left(\mathrm{Rec}Z.\sigma(\vec{X},Z;\vec{Y})\right)[\vec{X} := \vec{\xi};\vec{Y} := \vec{\eta}] :=$
$\mathrm{Rec}Z.\left(\sigma[\vec{X} := \vec{\xi}(\vec{X'},Z), Z := Z;\vec{Y} := \vec{\eta}]\right)$

[10]The rule (Arr) implicitely defines *positive and negative appearance*: A variable is positive in a type $\sigma \to \tau$ iff it is positive in τ and negative in σ and vice versa. All other rules preserve positivity and negativity.

Nothing changes for terms, syntactic values and the operational semantics, but we have to extend our definition of semantic values and our lemmata showing monotonicity, because we need the property

$$[\![(\sigma \to \tau)(\vec{X};\vec{Y})]\!]_{\vec{V};[\![\vec{\eta}]\!]} \subseteq [\![(\sigma \to \tau)(\vec{\xi};\vec{\eta})]\!]$$

for $\vec{V} \subseteq \mathrm{Val}^{\vec{\xi}}$ in order to show wellfoundedness.

Semantic Values. Be $\vec{V} \subseteq \mathrm{Val}^{\vec{\xi}}$, $\vec{W} \subseteq \mathrm{Val}^{\vec{\eta}}$. We define the set of semantic values $[\![\sigma(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}} \subseteq \mathrm{Val}^{\sigma(\vec{\xi};\vec{\eta})}$ by recursion over σ:

(Unit) $[\![1]\!] := \{()\}$

(Var) $[\![X_n]\!]_{\vec{V}} := V_n$

(Weak⁺) $[\![\sigma(\vec{X},X;\vec{Y})]\!]_{\vec{V},V;\vec{W}} := [\![\sigma(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}$

(Weak⁻) $[\![\sigma(\vec{X};\vec{Y},Y)]\!]_{\vec{V};\vec{W},W} := [\![\sigma(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}$

(Sum) $[\![(\sigma+\tau)(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}} := \{\mathrm{inl}(v) : v \in [\![\sigma(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}\} \cup \{\mathrm{inr}(v) : v \in [\![\tau(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}\}$

(Prod) $[\![(\sigma\times\tau)(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}} := \{(v,w) : v \in [\![\sigma(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}},\ w \in [\![\tau(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}\}$

(Arr) $[\![(\sigma\to\tau)(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}} := \{f \in \mathrm{Val}^{(\sigma\to\tau)(\vec{\xi};\vec{\eta})} : \forall u \in [\![\sigma(\vec{Y};\vec{X})]\!]_{\vec{W};\vec{V}}.\ \exists v \in [\![\tau(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}.\ f@u \Downarrow v\}$

(Rec) With the help of the abbreviation

$$\begin{aligned}\mathcal{F} \ :\ & \mathcal{P}\left(\mathrm{Val}^{\mathrm{Rec}\,X.\sigma(\vec{\xi},X;\vec{\eta})}\right) \to \mathcal{P}\left(\mathrm{Val}^{\mathrm{Rec}\,X.\sigma(\vec{\xi},X;\vec{\eta})}\right)\\ & V \mapsto \mathrm{fold}\left([\![\sigma(\vec{X},X;\vec{Y})]\!]_{\vec{V},V;\vec{W}}\right)\end{aligned}$$

we define $[\![\mathrm{Rec}\,X.\sigma(\vec{X},X;\vec{Y})]\!]_{\vec{V};\vec{W}} := \mathrm{lfp}\,\mathcal{F}$

We extend the lemma 4 by antitonicity for negative variables:

Lemma 17 (Monotonicity of semantic values)

(i) $\forall\sigma \in \mathrm{Ty}(\vec{X},Z,\vec{X}';\vec{Y}).\ A \subseteq B \to [\![\sigma]\!]_{\vec{V},A,\vec{V}';\vec{W}} \subseteq [\![\sigma]\!]_{\vec{V},B,\vec{V}';\vec{W}}$

(ii) $\forall\sigma \in \mathrm{Ty}(\vec{X};\vec{Y},Z,\vec{Y}').\ A \subseteq B \to [\![\sigma]\!]_{\vec{V};\vec{W},A,\vec{W}'} \supseteq [\![\sigma]\!]_{\vec{V};\vec{W},B,\vec{W}'}$

Proof simultaneously by induction over σ. Since most parts of this proof can be taken from proof of lemma 4 with small adjustments on the syntax, we only show the interesting cases (Arr) and (Rec)-(ii).

(Arr) (i) Assume an arbitrary $f \in [\![\sigma \to \tau]\!]_{\vec{V},A,\vec{V'};\vec{W}}$ (*) and a value $u \in [\![\sigma(\vec{Y};\vec{X},Z,\vec{X'})]\!]_{\vec{W};\vec{V},B,\vec{V'}}$. By induction hypothesis (ii) we see $u \in [\![\sigma]\!]_{\vec{W};\vec{V},A,\vec{V'}}$ and therefore since (*) exists $v \in [\![\tau]\!]_{\vec{V},A,\vec{V'};\vec{W}}$ such that $f@u \Downarrow v$. By induction hypothesis (i) we have $v \in [\![\tau]\!]_{\vec{V},B,\vec{V'};\vec{W}}$ and hence $f \in [\![\sigma \to \tau]\!]_{\vec{V},B,\vec{V'};\vec{W}}$.
(ii) Here we assume $f \in [\![\sigma \to \tau]\!]_{\vec{V};\vec{W},B,\vec{W'}}$ and show $f \in [\![\sigma \to \tau]\!]_{\vec{V};\vec{W},A,\vec{W'}}$. Proof analogous to (i).

(Rec) (i) Proof as in lemma 4.
(ii) Analogous to (i).
We define $\mathcal{F}'_{\_}(W) := [\![\sigma(\vec{X};\vec{Y},Z,\vec{Y'},Z')]\!]_{\vec{V};\vec{W},\_,\vec{W'},W}$ and $\mathcal{F}_{\_}(W) := \mathrm{fold}(\mathcal{F}'_{\_}(W))$. Thus we have to show $\mathrm{lfp}\,\mathcal{F}_B \subseteq \mathrm{lfp}\,\mathcal{F}_A$. We instantiate the induction hypothesis (ii) $\forall W.\mathcal{F}'_B(W) \subseteq \mathcal{F}'_A(W)$ for $\mathrm{lfp}\,\mathcal{F}_A$ and get by monotonicity of fold and by (ispfp) $\mathcal{F}_B(\mathrm{lfp}\,\mathcal{F}_A) \subseteq \mathcal{F}_A(\mathrm{lfp}\,\mathcal{F}_A) \subseteq \mathrm{lfp}\,\mathcal{F}_A$. Further by (ismpfp) $\mathrm{lfp}\,\mathcal{F}_B \subseteq \mathrm{lfp}\,\mathcal{F}_A$. □

Lemma 6 can be expanded to negative variables as well:

Lemma 18 (Substitution in semantic values)

$$\text{(i)} \quad [\![\sigma(\vec{X},Z,\vec{X'};\vec{Y})]\!]_{\vec{V},[\![\rho]\!],\vec{V'};\vec{W}} = [\![\sigma(\vec{X},\rho,\vec{X'};\vec{Y})]\!]_{\vec{V},\vec{V'};\vec{W}}$$
$$\text{(ii)} \quad [\![\sigma(\vec{X};\vec{Y},Z,\vec{Y'})]\!]_{\vec{V};\vec{W},[\![\rho]\!],\vec{W'}} = [\![\sigma(\vec{X};\vec{Y},\rho,\vec{Y'})]\!]_{\vec{V};\vec{W},\vec{W'}}$$

Proof simultaneously by induction over σ. Again most cases are trivial or shown as in lemma 6. We show case (Arr) exemplarily:

(Arr) (i) $f \in [\![(\sigma \to \tau)(\vec{X},Z,\vec{X'};\vec{Y})]\!]_{\vec{V},[\![\rho]\!],\vec{V'};\vec{W}}$ iff for all $u \in [\![\sigma(\vec{Y};\vec{X},Z,\vec{X'})]\!]_{\vec{W};\vec{V},[\![\rho]\!],\vec{V'}}$ there is a $v \in [\![\tau(\vec{X},Z,\vec{X'};\vec{Y})]\!]_{\vec{V},[\![\rho]\!],\vec{V'};\vec{W}}$ such that $f@u \Downarrow v$. Applying both induction hypotheses shows that this condition is equivalent to $\forall u \in [\![\sigma(\vec{Y};\vec{X},\rho,\vec{X'})]\!]_{\vec{W};\vec{V},\vec{V'}} \exists v \in [\![\tau(\vec{X},\rho,\vec{X'};\vec{Y})]\!]_{\vec{V},\vec{V'};\vec{W}}.\, f@u \Downarrow v$, which is the case iff $f \in [\![(\sigma \to \tau)(\vec{X},\rho,\vec{X'};\vec{Y})]\!]_{\vec{V},\vec{V'};\vec{W}}$.
(ii) analogously □

By iterated application of these two lemmata we get

Corollary 19 (Subset property of semantic values) *For all* σ, $\vec{V} \subseteq \mathrm{Val}^{\vec{\xi}}$ *we have*

$$[\![\sigma(\vec{X};\vec{Y})]\!]_{\vec{V};[\![\eta]\!]} \subseteq [\![\sigma(\vec{\xi};\vec{\eta})]\!]$$

In principle this corollary does hardly differ from the version for strictly positive type, since by lemma 18 it is equivalent to

$$\forall \sigma, \vec{\xi}, \vec{\eta}, \vec{V} \subseteq \mathrm{Val}^{\vec{\xi}}.\ [\![\sigma(\vec{X};\vec{\eta})]\!]_{\vec{V}} \subseteq [\![\sigma(\vec{\xi};\vec{\eta})]\!],$$

which again can be formulated as

$$\forall \sigma, \vec{\xi}, \vec{V} \subseteq \mathrm{Val}^{\vec{\xi}}.\ [\![\sigma(\vec{X};\emptyset)]\!]_{\vec{V};\emptyset} \subseteq [\![\sigma(\vec{\xi};\emptyset)]\!].$$

A slight modification has to be done on the definition of the

Codomain. For all functions $f \in [\![(\sigma \to \tau)(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}}$ we define

$$\mathrm{CoDom}(f) := \left\{ v \in [\![\tau(\vec{X};\vec{Y})]\!]_{\vec{V};\vec{W}} : \exists u \in [\![\sigma(\vec{Y};\vec{X})]\!]_{\vec{W};\vec{V}}.\ f@u \Downarrow v \right\}$$

Now since no change has to be done on the definition of the structural ordering and the accessibility propositions 10, 11 and 12, we can prove a reformulation of our central lemma 13. In fact as for corollary 19 the formulation will hardly differ from the original, since we can and need to show it only for types with no free negative variables. (We have to show it for types with free positive variables because of the recursive types.)

Lemma 20 (All semantic values are accessible)

$$\forall \sigma(\vec{X};\emptyset), \vec{\xi}.\ [\![\sigma(\vec{X};\emptyset)]\!]_{\mathrm{Acc}^{\vec{\xi}};\emptyset} \subseteq \mathrm{Acc}^{\sigma(\vec{\xi};\emptyset)}$$

An equivalent formulation would be

$$\forall \sigma(\vec{X};\vec{Y}), \vec{\xi}, \vec{\eta}.\ [\![\sigma(\vec{X};\vec{Y})]\!]_{\mathrm{Acc}^{\vec{\xi}},[\![\vec{\eta}]\!]} \subseteq \mathrm{Acc}^{\sigma(\vec{\xi};\vec{\eta})}$$

Proof by induction over $\sigma(\vec{X};\emptyset)$. Case (Weak$^-$) is not applicable, and all other cases can be treated exactly the same as in the proof of the original lemma, even case (Arr):

(Arr) Be $f \in [\![(\sigma \to \tau)(\vec{X};\emptyset)]\!]_{\mathrm{Acc}^{\vec{\xi}}}$. By IH we have $\mathrm{CoDom}(f) \subseteq [\![\tau(\vec{X};\emptyset)]\!]_{\mathrm{Acc}^{\vec{\xi}};\emptyset} \subseteq \mathrm{Acc}^{\tau(\vec{\xi};\emptyset)}$. By Corollary 19 we get $f \in [\![(\sigma \to \tau)(\vec{\xi};\emptyset)]\!]$ and hence by (accarr) $f \in \mathrm{Acc}^{(\sigma\to\tau)(\vec{\xi};\emptyset)}$.

In proving (Arr) we have no induction hypothesis for $\sigma(\emptyset;\vec{X})$, but we actually do not need one, because the accessibility of f depends only on the accessibility of the codomain and not of the domain. $\square$

7.2 Functions with multiple parameters

So far we have only considered functions that are structurally recursive by their first parameter. Now we want to extend our termination criterium to functions of multiple parameters. The termination checker foetus supports them already and calculates a permutation of the arguments such that the lexical ordering on them guarantees termination. In foetus the parameters have to be curried, but here we want to consider functions on tupels $\vec{v}$ of parameters because it fits easier in our formalization and formulation of strucural recursion. The proposition that structurally recursive function terminate is changed only slighty to

$$\frac{\forall \vec{v} \in [\![\sigma]\!].\ (\forall [\![\sigma]\!] \ni \vec{w} \prec \vec{v}.\ \langle \mathrm{rec}\, g.t; e\rangle @\vec{w} \Downarrow) \to \langle \mathrm{rec}\, g.t; e\rangle @\vec{v} \Downarrow}{\forall \vec{v} \in [\![\sigma]\!].\ \langle \mathrm{rec}\, g.t; e\rangle @\vec{v} \Downarrow}$$

Therefore all we have to do is do define the lexical ordering on products and prove that it is wellfounded.

Finite products. We define the nonempty finite product $\prod_{i=1}^{n} \sigma_i$, $n \geq 1$ of types σ_i as abbreviation by

$$\begin{aligned} \prod_{i=1}^{1} \sigma_i &:= \sigma_1 \\ \prod_{i=1}^{n+1} \sigma_i &:= \sigma_{n+1} \times \prod_{i=1}^{n} \sigma_i \quad (n \geq 1) \end{aligned}$$

Lexical ordering. Given the closed types $\sigma_1, \ldots, \sigma_n$ we inductively define $\prec_{(\sigma_n;\ldots;\sigma_1)} \subseteq [\![\prod_{i=1}^{n} \sigma_i]\!] \times [\![\prod_{i=1}^{n} \sigma_i]\!]$ as follows:

$$\text{(lexin)} \quad \frac{w <_{\sigma_1,\sigma_1} v}{w \prec_{(\sigma_1)} v}$$

$$\text{(lexlt)} \quad \frac{w <_{\sigma_{n+1},\sigma_{n+1}} v \quad \vec{w}, \vec{v} \in [\![\prod_{i=1}^{n} \sigma_i]\!]}{(w, \vec{w}) \prec_{(\sigma_{n+1};\ldots;\sigma_1)} (v, \vec{v})}$$

$$\text{(lexle)} \quad \frac{w \leq_{\sigma_{n+1},\sigma_{n+1}} v \quad \vec{w} \prec_{(\sigma_n;\ldots;\sigma_1)} \vec{v}}{(w, \vec{w}) \prec_{(\sigma_{n+1};\ldots;\sigma_1)} (v, \vec{v})}$$

For simplicity, unlike foetus we do not allow permutation of the components.

The accessible set $\mathrm{Acc}_{\prec}^{(\sigma_n;\ldots;\sigma_1)} \subseteq [\![\prod_{i=1}^n \sigma_i]\!]$ of the lexical ordering $\prec_{(\sigma_n;\ldots;\sigma_1)}$ is defined inductively as usual (we omit the index of $\prec$ where it is not necessary):

$$\text{(acclex)} \qquad \frac{\forall [\![\prod_{i=1}^n \sigma_i]\!] \ni \vec{w} \prec \vec{v}.\, \vec{w} \in \mathrm{Acc}_{\prec}^{(\sigma_n;\ldots;\sigma_1)}}{\vec{v} \in \mathrm{Acc}_{\prec}^{(\sigma_n;\ldots;\sigma_1)}}$$

Now based on the wellfoundedness of the structural ordering $<$ we can prove the

Lemma 21 (Wellfoundedness of the lexical ordering)

$$\forall \sigma_1, \ldots, \sigma_n.\, [\![\prod_{i=1}^n \sigma_i]\!] \subseteq \mathrm{Acc}_{\prec}^{(\sigma_n;\ldots;\sigma_1)}$$

Proof by induction on n. It demonstrates the proper use of wellfounded induction.

$n = 1$: We have to show $[\![\sigma_1]\!] \subseteq \mathrm{Acc}_{\prec}^{(\sigma_1)}$. Since by lemma 13 $[\![\sigma_1]\!] \subseteq \mathrm{Acc}^{\sigma_1}$ our goal follows from the formally stronger claim $\forall v \in \mathrm{Acc}^{\sigma_1}.v \in \mathrm{Acc}_{\prec}^{(\sigma_1)}$. Using the wellfounded induction principle (accind) we get the hypothesis (H) $\forall w < v.w \in \mathrm{Acc}_{\prec}^{(\sigma_1)}$ and have to show $v \in \mathrm{Acc}_{\prec}^{(\sigma_1)}$ which by (acc) is equivalent to $\forall w \prec v.\, w \in \mathrm{Acc}_{\prec}^{(\sigma_1)}$. Now the only matching case for $w \prec v$ is (lexin), therefore $w < v$ and we can finish this part of the proof by (H).

$n \to n{+}1$: Our goal is $\forall v \in [\![\sigma_{n+1}]\!], \vec{v} \in [\![\prod_{i=1}^n \sigma_i]\!].\, (v, \vec{v}) \in \mathrm{Acc}_{\prec}^{(\sigma_{n+1};\ldots;\sigma_1)}$ (in the following abbreviated to $\mathrm{Acc}_{\prec}$). Again, since $[\![\sigma_{n+1}]\!] \subseteq \mathrm{Acc}^{\sigma_{n+1}}$, we get the hypothesis (H1) $\forall w < v, \vec{v} \in [\![\prod_{i=1}^n \sigma_i]\!].\, (w, \vec{v}) \in \mathrm{Acc}_{\prec}$ by (accind). Now in the same way, since by induction hypothesis $[\![\prod_{i=1}^n \sigma_i]\!] \subseteq \mathrm{Acc}_{\prec}^{(\sigma_n;\ldots;\sigma_1)}$, we can apply the wellfounded induction principle for the lexical order and get a second hypothesis (H2) $\forall \vec{w} \prec \vec{v}.\, (v, \vec{w}) \in \mathrm{Acc}_{\prec}$ to show the remaining goal $(v, \vec{v}) \in \mathrm{Acc}_{\prec}$. This follows from $\forall (w, \vec{w}) \prec (v, \vec{v}).\, (w, \vec{w}) \in \mathrm{Acc}_{\prec}$. Now $(w, \vec{w}) \prec (v, \vec{v})$ can be generated in two ways: Case (lexlt) $w < v$: We finish our proof by (H1). Case (lexle) $w \leq v$, $\vec{w} \prec \vec{v}$: By (H2) we get $(v, \vec{w}) \in \mathrm{Acc}_{\prec}$ and by the following small lemma also $(w, \vec{w}) \in \mathrm{Acc}_{\prec}$. $\square$

Lemma 22

$$\forall (v, \vec{v}) \in \mathrm{Acc}_{\prec}, v' \leq v.\, (v', \vec{v}) \in \mathrm{Acc}_{\prec}$$

Proof. We have to prove $\forall (w,\vec{w}) \prec (v',\vec{v}).\,(w,\vec{w}) \in \mathrm{Acc}_{\prec}$. If we can show $(w,\vec{w}) \prec (v',\vec{v}) \to (w,\vec{w}) \prec (v,\vec{v})$, then the rest follows by (acclex^{-1}). Case analysis on $(w,\vec{w}) \prec (v',\vec{v})$:

(lexlt) $w < v'$: By transitivity of the structural ordering we get $w < v$ and hence by (lexlt) $(w,\vec{w}) \prec (v,\vec{v})$.

(lexlt) $w \leq v'$, $\vec{w} \prec \vec{v}$: Again by transitivity of the structural ordering we get $w \leq v$ and hence by (lexle) $(w,\vec{w}) \prec (v,\vec{v})$. □

Now having proven wellfoundedness of the lexical ordering we can redefine the set of structurally recursive terms to include the functions the termination of which is ensured by the lexical ordering ($\sigma := \prod_{i=1}^{n} \sigma_i$):

$$\begin{array}{rcl} \mathrm{SR}^{\sigma\to\tau}[\Gamma] & := & \{\mathrm{rec}\, g.t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma] : \forall e \in [\![\Gamma]\!], \vec{v} \in [\![\sigma]\!]. \\ & & (\forall \vec{w} \prec_{(\sigma_n;\ldots;\sigma_1)} \vec{v}.\langle \mathrm{rec}\, g.t; e\rangle @\vec{w} \Downarrow) \to \langle \mathrm{rec}\, g.t; e\rangle @\vec{v} \Downarrow\} \end{array}$$

By wellfounded induction for the lexical ordering we show that these structurally functions terminate on all inputs (as in lemma 15), and then by defining the good terms with the new definition of SR we can prove normalisation without further changes.

7.3 Mutual recursion

The termination checker foetus also supports mutual recursive functions. Our term syntax so far only allows to define one recursive function at a time, thus mutual recursion is not possible:

$$(\mathrm{rec})\quad \frac{t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma, g^{\sigma\to\tau}]}{\mathrm{rec}\, g.t \in \mathrm{Tm}^{\sigma\to\tau}[\Gamma]}$$

$$(\mathrm{oprec})\quad \frac{}{\langle \mathrm{rec}\, g.t; e\rangle \Downarrow \langle \mathrm{rec}\, g.t; e\rangle}$$

$$(\mathrm{opappvr})\quad \frac{\langle t; e, g = \mathrm{rec}\, g.t\rangle \Downarrow f \quad f@u \Downarrow v}{\langle \mathrm{rec}\, g.t; e\rangle @u \Downarrow v}$$

But these rules can easily be adopted for mutual recursion. We introduce the new term formers rec_i^n ($1 \leq i \leq n$) where rec_i^n denotes the ith of n simultaneous defined recursive functions. For each n we get the n introduction rules (written as one with n conclusions):

$$(\mathrm{rec}^n)\quad \frac{\forall 1 \leq i \leq n.\ t_i \in \mathrm{Tm}^{\sigma_i\to\tau_i}[\Gamma, g^{\sigma_1\to\tau_1}, \ldots, g^{\sigma_n\to\tau_n}]}{\forall 1 \leq i \leq n.\ \mathrm{rec}_i^n(g_1 = t_1, \ldots, g_n = t_n) \in \mathrm{Tm}^{\sigma_i\to\tau_i}[\Gamma]}$$

The operational sematics of course is extended by the following rules (we abbreviate $g_1 = t_1, \ldots, g_n = t_n$ to $\vec{g} = \vec{t}$):

$$(\mathrm{oprec}_i^n) \qquad \frac{}{\langle \mathrm{rec}_i^n(\vec{g}=\vec{t}); e\rangle \Downarrow \langle \mathrm{rec}_i^n(\vec{g}=\vec{t}); e\rangle} \text{ for all } n,\ 1 \leq i \leq n.$$

$$(\mathrm{opappvr}_i^n) \qquad \frac{\langle t_i; e, g_1 = \mathrm{rec}_1^n(\vec{g}=\vec{t}), \ldots, g_n = \mathrm{rec}_n^n(\vec{g}=\vec{t})\rangle \Downarrow f \quad f@u \Downarrow v}{\langle \mathrm{rec}_i^n(\vec{g}=\vec{t}); e\rangle @u \Downarrow v}$$
for all n, $1 \leq i \leq n$.

Now a recursive function $\mathrm{rec}_i^n(\vec{g} = \vec{t})$ is structurally recursive only if all of the simultaneous defined functions that are called directly or indirectly are structurally recursive as well. In [Abe98]we explain an algorithm to decide structural recursiveness for mutual recursive functions with perhaps multiple parameters in detail.

For our soundness proof nothing changes since we assume structural recursiveness.

8 Conclusion

In this work we have shown the soundness of our method to ensure a recursive function terminates on all inputs. We have confirmed our assumption that from structural recursiveness we can infer totality. Of course it would be desirable to extend our system further to polymorphic and dependent types.

But what is still missing is the syntactical aspect: It remains to show that indeed structurally recursive functions are the semantics of structurally recursive *terms*, i.e. terms that contain recursive calls only with smaller arguments w.r.t. a structural ordering on *terms*. (In section 6 we have defined the set SR^{τ} of terms, but their property of structural recursiveness has been ensured by means of semantics.) This syntactical check now is implemented in foetus, and it would be worthwhile formulating it as a proof, from which we could again extract a termination checker. (From the proof in this work we can extract nothing, since wellfoundedness proofs have no computational content.)

During the production of this work I encountered an application of it. I formulated parts of my system in LEGO and did some of the accessibility proofs by pattern matching. But these proofs are only sound if the patterns are total and structurally recursive. Unfortunately, LEGO contains no checker like foetus so far, and thus I could not be sure if my proofs were correct.

I think that in the area of machine checked proofs the pattern-matching-style proofs will replace elimination-style proofs, since once one has adjusted oneself to this new way of thinking, one can prove more intuitively. (It is like programming: no practically minded person would prefer a programming language with only primitive recursion to a full-featured functional programming language like SML or Haskell.) And thus termination checkers for the theorem provers will be implemented as Thorsten Altenkirch and I have done, and for example Catarina Coquand is doing for ALF at the University of Göteborg.

References

[Abe98] Andreas Abel. foetus – termination checker for simple functional programs. http://www.informatik.uni-muenchen.de/~abel/foetus/, 1998.

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A user's guide to ALF*. Department of Computing Science, University of Göteborg/Chalmers, http://www.cs.chalmers.se/Cs/Research/Logic/alf/guide.html, May 1994.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer Verlag, 1993.

[BG96] Jürgen Brauburger and Jürgen Giesl. Termination analysis for partial functions. *In Proceedings of the Third International Static Analysis Symposium (SAS'96), Aachen, Germany, Lecture Notes in Computer Science 1145, Springer-Verlag*, 1996.

[Coq92] Thierry Coquand. Pattern matching with dependent types. *(to be updated)*, 1992.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–115, 1987.

[Gie97] Jürgen Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning 19: 1–29*, 1997.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.

[HPF97] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell, Version 1.4*. Yale Haskell Project, http://www.haskell.org/tutorial/, March 1997.

[LP92] Zhaohui Luo and Robert Pollack. Lego proof development system: User's manual. Lfcs, Computer Science Department, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Scotland, May 1992. Updated version. See http://www.dcs.ed.ac.uk/home/lego.

[Mat98] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types.* PhD thesis, Ludwig-Maximilian-University, 1998.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Ste95] J. Steinbach. Simplification orderings: History of results. *Fundamenta Informaticae*, 24:47–87, 1995.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.