Specification and Verification of a Formal System for Structurally Recursive Functions

Andreas Abel

Department of Computer Science, University of Munich
abel@informatik.uni-muenchen.de

Abstract. A type theoretic programming language is introduced that is based on lambda calculus with coproducts, products and inductive types, and additionally allows the definition of recursive functions in the way that is common in most functional programming languages. A formal system is presented that checks whether such a definition is structurally recursive and a soundness theorem is shown for this system. Thus all functions passing this check are ensured to terminate on all inputs. For the moment only non-mutual recursive functions are considered.

1 Introduction

In lambda calculi with inductive types the standard means to construct a function over an inductive type is the *recursor*. This method, however, has several drawbacks, as discussed, e.g., in [Coq92]. One of them is that it is not very intuitive: For instance it is not obvious how to code the following "division by 2"-function with recursors:

$$\begin{aligned}
&\text{half } 0 &&= 0\\
&\text{half } 1 &&= 0\\
&\text{half } \mathrm{n}{+}2 &&= (\text{half } \mathrm{n}){+}1
\end{aligned}$$

Therefore an alternative approach has been investigated: *recursive definitions with pattern matching*, as they are common in nearly all functional programming languages. To define total functions, they have to satisfy two conditions:

1. The patterns have to be exhaustive and mutually exclusive. We will not focus on this point further since the foetus language we introduce in Sect. 2 uses only case expressions and thus this condition is always fulfilled. For a discussion see [Coq92].
2. The definition must be well-founded, which means that for all arguments the function value has a well-founded computation tree. This can be ensured if one can give a *termination ordering* for that function, i.e., a ordering with respect to which its arguments in each recursive call are smaller than the input parameters of that function.

We will restrict ourselves to *structural orderings*, since they can be checked automatically and still allow the definition of a large number of total functions directly. By introducing an extra wellfounded argument (which is decreased structurally in each recursive call), one can code any function that can be proven total in Martin-Löf's Type Theory (MLTT [Mar84]) or an equivalent system.

We say an expression s is structurally smaller than an expression t if s appears in the computation tree of t. Sometimes this is called "component relation".

In [Abe98] the implementation of the termination checker foetus has been presented. This checker, which accepts structurally recursive functions of the kind described above, has been reimplemented as part of Agda by C. Coquand [Coq99]. In [AA99] a semantics for the foetus language has been defined, and for all types the wellfoundedness wrt. the structural ordering on values has been proven. Furthermore a function has been defined to be *(semantically)* structurally recursive if it terminates on input value w under the condition that it terminates on all values v that are smaller than w wrt. a structural ordering on the *value* domain. Thus we could prove termination for all terms by assuming that all named functions are structurally recursive.

This article is meant to close a gap that has remained in [AA99]. For it to be self-contained, we repeat the definitions of the foetus language, operational and value semantics as far as we refer to them in this presentation. (Old definitions are laid out in tables.) The new contributions start with a formalization of the termination checker in the form of a predicate "sr" (*syntactically* structurally recursive) on terms. This extends the derivation system for structural ordering on *terms*. First we will show that the ordering on terms is reflected by that on the values, i.e., evaluation preserves the ordering. Second we will prove that all functions captured by the sr-predicate are indeed total, i.e., that they terminate on all inputs. Thus we establish the soundness of our system formally.

At the moment we exclude mutually recursive functions, since they require more sophisticated concepts. For non-mutual recursion—which we will treat here—the proof is beautiful in its straightforwardness. The specification consists mostly of strictly positive inductive definitions, and the proof is constructive. Also, since most details have been completely formalized, we can almost directly implement it in a system like MLTT. On the final theorem, "all closed terms evaluate to a good value", we can apply program extraction. Thus we would obtain an interpreter for the lambda calculus with inductive types and recursive functions, for which termination is verified.

1.1 Related Work

Giménez also presents a syntactic criterion for structural recursion which he calls "guarded-by-destructors" [Gim95]. He gives, however, no proof for the soundness of his criterion. Furthermore we believe that our approach is more concise and more flexible in how functions can be defined.

Jouannaud, Okada [JO97] and later also Blanqui [BJO00] deal with inductive types, too, but in the area of extensible term rewriting systems. Since they also do not handle mutual recursion, our present approach seems to have the same expressive power than their *Extended General Schema*. But both approaches differ considerably in the notion of a recursive call with a smaller argument.

Telford and Turner [TT99] are investigating *Strong Functional Programming*, i.e., functional programming languages where only terminating functions can be defined. Technically they use *abstract interpretations* to ensure termination. Since they count constructors and destructors, they can handle a wider class of functions. I consider their approach as very promising but it seems that it is not yet fully formalized and verified. Maybe the ideas of my work presented here could be transferred to their approach.

Christoph Walther has invented *reduction checking*, which is sometimes referred to as "Walther recursion" [MA96]. Here functions are not only examined whether they are terminating, but also whether they are reducers or preservers, i.e., whether the output of the function is (strictly) smaller than the input. This information can be used to check termination of nested recursive functions or just for functions which decrease their input via a previously defined function in a recursive call. It seems that my system could be extended to reduction checking in a straightforward way, since I already use assumptions (dependencies) in my judgements (see Def. 5).

Finally, Pientka and Pfenning [PP00] have implemented termination and reduction checking for the Twelf system [PS98] based on LF [HHP93]. Although coming from the realm of logic programming, their formal system that implements the check is similar to mine. However, theirs is *constructor* based and mine is *destructor* based. They justify the stringency of their system by a cut admissibility proof, but have not shown soundness yet.

In comparision with the work discussed above, the main contribution of this article is giving a clear and straightforward *soundness* proof for my system, based on a *semantics* for the judgements of my formal system.

1.2 Notational conventions

We are using vectors to simplify our notation. If we have a family of (meta) expressions $E_1, E_2, \ldots, E_n$ we write $\boldsymbol{E}$ for the whole sequence. S_n denotes the set of permutations on $\{1, \ldots, n\}$. Furthermore we use the denotations

X, Y, Z	for	type variables
ρ, σ, τ	for	types
g, x, y, z	for	term variables
r, s, t, a	for	terms
f, u, v, w	for	values
c	for	closures
e	for	environments
p, q	for	atoms (containing a relation between terms)

We use rule notation for inductive definitions, but also for propositions (cf. Lemma 1). On top of the bar we put the premises and on bottom the conclusion(s), in both cases to be read as a conjunction.

2 The foetus Language and its Semantics

In [AA99] we already introduced the term language foetus and its semantics. Here we will only briefly repeat the definitions and main results (see Tables 2 and 3). The types are constructed by type variables $X, Y, Z, \ldots$ and the type constructors Σ (finite sum), Π (finite product), $\rightarrow$ (function space) and μ (strictly positive inductive type).

Types $\mathsf{Ty}(\boldsymbol{X})$ (over a finite list of type variables $\boldsymbol{X}$)

$\tau, \sigma, \boldsymbol{\sigma}$	$::= X_i \mid \Sigma\boldsymbol{\sigma} \mid \Pi\boldsymbol{\sigma} \mid \sigma \rightarrow \tau \mid \mu X_n.\sigma$	precedence: $\Pi, \Sigma, \rightarrow, \mu$
0	$:= \Sigma()$	empty sum
1	$:= \Pi()$	empty product
Ty	$:= \mathsf{Ty}()$	closed types

Contexts

$$\Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \in \mathsf{Cxt} \qquad x_i \text{pairwise distinct}$$

Terms $\mathsf{Tm}^\sigma[\Gamma]$ of closed type σ in context Γ

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

$$(\text{in})\ \frac{t \in \mathsf{Tm}^{\sigma_j}[\Gamma] \qquad \boldsymbol{\sigma} \in \mathsf{Ty}}{\text{in}_j(t) \in \mathsf{Tm}^{\Sigma\boldsymbol{\sigma}}[\Gamma]} \qquad (\text{case})\ \frac{t \in \mathsf{Tm}^{\Sigma\boldsymbol{\sigma}}[\Gamma] \qquad t_i \in \mathsf{Tm}^\rho[\Gamma, x_i^{\sigma_i}]}{\text{case}(t, x_1^{\sigma_1}.t_1, \ldots, x_n^{\sigma_n}.t_n) \in \mathsf{Tm}^\rho[\Gamma]}$$

$$(\text{tup})\ \frac{t_i \in \mathsf{Tm}^{\sigma_i}[\Gamma] \text{ for } 1 \leq i \leq n}{(t_1, \ldots, t_n) \in \mathsf{Tm}^{\Pi\boldsymbol{\sigma}}[\Gamma]} \qquad (\text{pi})\ \frac{t \in \mathsf{Tm}^{\Pi\boldsymbol{\sigma}}[\Gamma]}{\text{pi}_j(t) \in \mathsf{Tm}^{\sigma_j}[\Gamma]}$$

$$(\text{lam})\ \frac{t \in \mathsf{Tm}^\tau[\Gamma, x^\sigma]}{\lambda x^\sigma.t \in \mathsf{Tm}^{\sigma\rightarrow\tau}[\Gamma]} \qquad (\text{rec})\ \frac{t \in \mathsf{Tm}^\tau[\Gamma, g^{\Pi\boldsymbol{\sigma}\rightarrow\tau}, x^{\Pi\boldsymbol{\sigma}}] \qquad \boxed{\vdash g(x) \ \mathsf{sr}\ t}}{\text{fun}\ g^{\Pi\boldsymbol{\sigma}\rightarrow\tau}(x^{\Pi\boldsymbol{\sigma}}) = t \in \mathsf{Tm}^{\Pi\boldsymbol{\sigma}\rightarrow\tau}[\Gamma]}$$

$$(\text{app})\ \frac{t \in \mathsf{Tm}^{\sigma\rightarrow\tau}[\Gamma] \qquad s \in \mathsf{Tm}^\sigma[\Gamma]}{t\, s \in \mathsf{Tm}^\tau[\Gamma]}$$

$$(\text{fold})\ \frac{t \in \mathsf{Tm}^{\sigma(\mu X.\sigma)}[\Gamma]}{\text{fold}(t) \in \mathsf{Tm}^{\mu X.\sigma}[\Gamma]} \qquad (\text{unfold})\ \frac{t \in \mathsf{Tm}^{\mu X.\sigma}[\Gamma]}{\text{unfold}(t) \in \mathsf{Tm}^{\sigma(\mu X.\sigma)}[\Gamma]}$$

Syntactic sugar for $\text{fun}\ g(y^{\Pi\boldsymbol{\sigma}}) = t[x_1 := \text{pi}_1(y), \ldots, x_n := \text{pi}_n(y)]$

$$\frac{t \in \mathsf{Tm}^\tau[\Gamma, g^{\Pi\boldsymbol{\sigma}\rightarrow\tau}, x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}] \qquad \vdash g(y)\ \mathsf{sr}\ t[x_1 := \text{pi}_1(y), \ldots, x_n := \text{pi}_n(y)]}{\text{fun}\ g(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}) = t \in \mathsf{Tm}^{\Pi\boldsymbol{\sigma}\rightarrow\tau}[\Gamma]}$$

Table 2. The foetus Language

2.1 Recursive Terms

The terms inhabiting *closed* types are the terms of a lambda calculus with sums and products plus folding and unfolding for inductive types plus structurally recursive terms (see Table 2). Here fold and unfold establish the isomorphism between $\mu X.\sigma$ and $\sigma(\mu X.\sigma)$. The rule (rec) introduces a recursive term resp. named function that solves the equation $g = \lambda x.t$ in place of g.[1] We require the argument of g to be of product type to simplify the handling of lexicographic termination orderings.[2]

In the *function body* t the identifier g may only appear structurally recursive with respect to parameter x. This property is ensured by the judgement

$$\Delta \vdash g(x) \ \mathsf{sr}\ t$$

Read "under the dependencies Δ the function g with parameter x is structurally recursive in t". For example the following judgement is valid

$$x' <^{\mathsf{Tm}} x \vdash g(x) \ \mathsf{sr}\ g\, x'$$

which expresses that a recursive call $g\, x'$ in a function g is admissible for the function $g(x)$ to be structurally recursive, if the argument x' of the call is strictly smaller than the parameter x of the function. A *function* g is structurally recursive, if it is structurally recursive in its whole body t under no assumptions (dependencies), i.e., if all calls are decreasing.

We will present the proof rules for $<^{\mathsf{Tm}}$ and sr in Sect. 3. The intention is that is defined simultaneously with the terms.

2.2 Operational Semantics

We define a big step operational semantics "$\downarrow$" as a relation between closures and values. The intention behind values is that they are the results of the evaluation process. Closures are (open) terms paired with a mapping of the free variables to values (environment). Closures of the form $f@u$ (value applied to values) are convenient to define the operational semantics without casting values back to terms.

Table 3 presents the operational semantics. (For reasons of readability we leave out type and context annotations wherever possible.) Our strategy is call-by-value (see rule (opapp)) and we do not evaluate under λ and recursive terms (see rules (oplam) and (oprec)). Furthermore $\downarrow$ is deterministic, i.e., $c \downarrow v$ and $c \downarrow v'$ imply $v = v'$. Thus we can invert all rules for $\downarrow$. We denote the inversion of the rule (X) by (X^{-1}).

---

[1] In the literature one often finds the notation $\mu g.\lambda x.t$, expressing that the function is the smallest fixed-point of $\lambda x.t[\Gamma, g]$. Our notation is inspired by the functional programming language SML.

[2] Note that 1-element tuples are allowed as arguments, so this is not really a restriction.

Values Val^σ, **environments** $\mathsf{Env}[\Gamma]$ and **closures** Cl^τ

$$
\begin{array}{rcll}
v, w, \boldsymbol{v} \in \mathsf{Val} & ::= & \mathrm{in}_j(v) \mid (v_1, \ldots, v_n) \mid \mathrm{fold}(v) \mid \langle \lambda x.t; e\rangle \mid \langle \mathrm{fun}\, g(x) = t; e\rangle & \\
\mathsf{Env}[\Gamma] & := & \{x_1 = v_1, \ldots, x_n = v_n : v_i \in \mathsf{Val}^{\sigma_i}\} & \Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \\
\mathsf{Cl}^\tau & := & \{\langle t; e\rangle : \exists \Gamma \in \mathsf{Cxt}.\ t \in \mathsf{Tm}^\tau[\Gamma] \wedge e \in \mathsf{Env}[\Gamma]\} & \\
& \cup & \{f@u : f \in \mathsf{Val}^{\sigma \to \tau}, u \in \mathsf{Val}^\sigma\} & (\text{“@” is a new symbol})
\end{array}
$$

Notation

terms: $t^\sigma[\Gamma]$ expresses t, where $t \in \mathsf{Tm}^\sigma[\Gamma]$ (σ, Γ optional)
values: v^σ expresses v, where $v \in \mathsf{Val}^\sigma$

Operational Semantics $\downarrow^\sigma \subseteq \mathsf{Cl}^\sigma \times \mathsf{Val}^\sigma$

$$
(\mathrm{opvar})\ \frac{}{\langle x; e, x = v\rangle \downarrow v}
\qquad
(\mathrm{opweak})\ \frac{\langle t[\Gamma]; e\rangle \downarrow v}{\langle t[\Gamma, x]; e, x = w\rangle \downarrow v}
\qquad
(\mathrm{opin})\ \frac{\langle t; e\rangle \downarrow v}{\langle \mathrm{in}_j(t); e\rangle \downarrow \mathrm{in}_j(v)}
$$

$$
(\mathrm{opcase})\ \frac{\langle t^{\Sigma \boldsymbol{\sigma}}[\Gamma]; e\rangle \downarrow \mathrm{in}_j(w^{\sigma_j}) \qquad \langle t_j^\tau[\Gamma, x_j^{\sigma_j}]; e, x_j = w\rangle \downarrow v^\tau}{\langle \mathrm{case}(t, \boldsymbol{x.t}); e\rangle \downarrow v}
$$

$$
(\mathrm{optup})\ \frac{\langle t_i; e\rangle \downarrow v_i \text{ for } 1 \le i \le n}{\langle (\boldsymbol{t}); e\rangle \downarrow (\boldsymbol{v})}
\qquad
(\mathrm{oppi})\ \frac{\langle t; e\rangle \downarrow (\boldsymbol{v})}{\langle \mathrm{pi}_j(t); e\rangle \downarrow v_j}
$$

$$
(\mathrm{oplam})\ \frac{}{\langle \lambda x.t; e\rangle \downarrow \langle \lambda x.t; e\rangle}
\qquad
(\mathrm{oprec})\ \frac{}{\langle \mathrm{fun}\, g(x) = t; e\rangle \downarrow \langle \mathrm{fun}\, g(x) = t; e\rangle}
$$

$$
(\mathrm{opapp})\ \frac{\langle t; e\rangle \downarrow f \qquad \langle s; e\rangle \downarrow u \qquad f@u \downarrow v}{\langle t\ s; e\rangle \downarrow v}
$$

$$
(\mathrm{opappvl})\ \frac{\langle t; e, x = u\rangle \downarrow v}{\langle \lambda x.t; e\rangle@u \downarrow v}
\qquad
(\mathrm{opappvr})\ \frac{\langle t; e, g = \langle \mathrm{fun}\, g(x) = t; e\rangle, x = u\rangle \downarrow v}{\langle \mathrm{fun}\, g(x) = t; e\rangle@u \downarrow v}
$$

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

Table 3. Values and Operational Semantics

2.3 Semantics and Structural Ordering on Values

We give a semantics of the types in foetus that captures the “good” values, i.e. these values that ensure termination. In this sense f will be a good function value if it evaluates to a good result if applied to a good argument.

Definition 1 (Semantics on values). *For every closed type* $\sigma \in \mathsf{Ty}$ *the semantics* $\mathsf{VAL}^\sigma \subseteq \mathsf{Val}^\sigma$ *contains the good values of type* σ*. Especially for arrow types:*

$$f \in \mathsf{VAL}^{\sigma \to \tau} \iff \forall u \in \mathsf{VAL}^\sigma.\ \exists v \in \mathsf{VAL}^\tau.\ f@u \downarrow v$$

More details do not matter at this point. We refer the interested reader to [Abe99], where the construction of this semantics has been carried out, and to

[AA99] for an even predicative construction. Note that in both papers VAL^σ is denoted with $[\![\sigma]\!]$.

Definition 2 (Good environment). *Let $\Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$. Then*

$$\mathsf{ENV}[\Gamma] := \{x_1 = v_n, \ldots, x_n = v_n : v_i \in \mathsf{VAL}^{\sigma_i} \text{ for } 1 \leq i \leq n\}$$

Definition 3 (Strong evaluation). *We say a closure $c \in \mathsf{Cl}^\sigma$ evaluates strongly to a value $v \in \mathsf{Val}^\sigma$, if v is a good value.*

$$c \Downarrow v :\Longleftrightarrow c \downarrow v \ \& \ v \in \mathsf{VAL}^\sigma$$

For closures of the form $c \equiv \langle t; e\rangle$ we additionally require that every subterm s of t evaluates strongly $\langle s; e'\rangle \Downarrow$ in environment e' where e' is e, a shortening of e or the extension of e by a good value w. We refer to this requirement as the subterm property*.*

Structural ordering $R_{\sigma,\tau} \subseteq \mathsf{VAL}^\sigma \times \mathsf{VAL}^\tau$ $(R \in \{<, \leq\})$

$$(\leq\text{refl})\ \frac{}{v \leq_{\sigma,\sigma} v} \qquad (R\text{in})\ \frac{w\ R_{\rho,\sigma_j}\ v}{w\ R_{\rho,\Sigma\boldsymbol{\sigma}}\ \text{in}_j(v)} \qquad (R\text{tup})\ \frac{\exists j.\ w\ R_{\rho,\sigma_j}\ v_j}{w\ R_{\rho,\Pi\boldsymbol{\sigma}}\ (\boldsymbol{v})}$$

$$(R\text{arr})\ \frac{f@u \Downarrow v \qquad w\ R_{\rho,\tau}\ v}{w\ R_{\rho,\sigma\to\tau}\ f} \qquad (R\text{fold})\ \frac{w \leq_{\sigma,\tau(\mu X.\tau)} v}{w\ R_{\sigma,\mu X.\tau}\ \text{fold}(v)}$$

Admissible rules (besides transitivity)

$$(\leq<)\ \frac{w <_{\rho,\tau} v}{w \leq_{\rho,\tau} v} \qquad (R\text{in'})\ \frac{\text{in}_j(v)\ R\ w}{v\ R\ w} \qquad (R\text{tup'})\ \frac{(\boldsymbol{v})\ R\ w}{v_j\ R\ w}$$

$$(R\text{arr'})\ \frac{f\ R\ w \qquad f@u \Downarrow v}{v\ R\ w} \qquad (R\text{fold'})\ \frac{\text{fold}(v) \leq w}{v\ R\ w}$$

Lexicographic ordering $\prec^k_{\pi,\boldsymbol{\sigma}} \subseteq \mathsf{VAL}^{\Pi\boldsymbol{\sigma}} \times \mathsf{VAL}^{\Pi\boldsymbol{\sigma}}$ for closed types $\boldsymbol{\sigma}$, a permutation $\pi \in S_n$ and $k \in \mathbb{N}$

$$(\text{lex}<)\ \frac{v_{\pi(k)} < w_{\pi(k)}}{(\boldsymbol{v}) \prec^k_{\pi,\boldsymbol{\sigma}} (\boldsymbol{w})} \qquad (\text{lex}\leq)\ \frac{v_{\pi(k)} \leq w_{\pi(k)} \qquad (\boldsymbol{v}) \prec^{k+1}_{\pi,\boldsymbol{\sigma}} (\boldsymbol{w})}{(\boldsymbol{v}) \prec^k_{\pi,\boldsymbol{\sigma}} (\boldsymbol{w})}$$

Table 4. Ordering on Values

Table 4 shows the definition of the lexicographic extension $\prec_\pi$ of the structural ordering on values, which we obtain from $\prec^k_\pi$ for $k = 1$. In [AA99] we proved that it is wellfounded. We will exploit this fact later in the proof of "f terminates at input v", doing noetherian induction. Thus we have the hypothesis for all smaller $v' \prec v$ at hand for the proof of termination at v.

3 A Formal System for Structural Recursion

In this section we will introduce the foetus termination calculus, but first motivate it by an example. Consider the following foetus implementation of the addition of ordinal numbers. We define two type abbreviations

$$\begin{aligned}\mathbf{Nat} &\equiv \mu X.1 + X\\ \mathbf{Ord} &\equiv \mu X.1 + X + (\mathbf{Nat} \to X)\end{aligned}$$

The constructors of **Ord** and the addition function are

$$\begin{aligned}\mathbf{O} &\equiv \mathrm{fold}(\mathrm{in}_1())\\ \mathbf{S}(v) &\equiv \mathrm{fold}(\mathrm{in}_2(v))\\ \mathbf{Lim}(f) &\equiv \mathrm{fold}(\mathrm{in}_3(f))\end{aligned}$$

$$\begin{aligned}\mathbf{add} \equiv \lambda x^{\mathbf{Ord}}.\ & \mathrm{fun\,add}^{\Pi(\mathbf{Ord})\to\mathbf{Ord}}(y^{\mathbf{Ord}}) =\\ & \mathrm{case}(\mathrm{unfold}(y),\\ & \qquad \_^{1}.\ x,\\ & \qquad n^{\mathbf{Ord}}.\ \mathbf{S}(\mathrm{add}(n)),\\ & \qquad f^{\mathbf{Nat}\to\mathbf{Ord}}.\ \mathbf{Lim}(\lambda z^{\mathbf{Nat}}.\ \mathrm{add}(f\,z)))\end{aligned}$$

(The superindex 1 in the first branch of the case expression is just a type annotation, stating that the variable _ can only contain the empty tuple.)

In our term language we can only define **add**, if add is structurally recursive in its function body. For this we require that in all recursive calls the argument is structurally smaller than the input parameter of the function. In our case this gives us the proof obligations

1. $n < y$
2. $f\,z < y$

Our approach works as follows: We descend into the function body until we reach the recursive calls, and on our way we collect dependency information between variables. These dependencies are generated whenever we pass a *case*-expression. Thus for call 1 we get the dependency $n \leq \mathrm{unfold}(y)$. From this we infer $n < y$ since we require a folding step to increase the structural ordering strictly.

For call 2 we infer $f\,z < y$ from $f \leq \mathrm{unfold}(y)$. We justify this by $f\,z \leq f$. The latter is valid since we regard functions as (possibly infinitely branching) trees and application as selection of one branch.

The formalization of the above informally described method consists of three relations on terms:

1. the structural ordering $<^{\mathsf{Tm}}$,
2. its lexicographic extension $\prec^{\mathsf{Tm}}$ (needed e.g. for the Ackermann function) and
3. the predicate of structural recursiveness "sr".

3.1 Structural Ordering on Terms

In the following we will make precise the definition of the structural ordering on terms and give rules that allow us to derive a relation between two terms under a given set of dependencies.

Right hand side rules ($R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}$):

$$(R\mathrm{caseR})\ \frac{\Delta, x_i \leq^{\mathsf{Tm}} s \vdash s_i \; R \; t \quad \text{for } i = 1, \ldots, n}{\Delta \vdash \mathrm{case}(s, \boldsymbol{x}.\boldsymbol{s}) \; R \; t} \qquad (R\mathrm{piR})\ \frac{\Delta \vdash s \; R \; t}{\Delta \vdash \mathrm{pi}_j(s) \; R \; t}$$

$$(R\mathrm{appR})\ \frac{\Delta \vdash s \; R \; t}{\Delta \vdash s \; a \; R \; t} \qquad (R\mathrm{unfR})\ \frac{\Delta \vdash s \leq^{\mathsf{Tm}} t}{\Delta \vdash \mathrm{unfold}(s) \; R \; t}$$

Left hand side rules ($R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}$):

$$(R\mathrm{caseL})\ \frac{\Delta, x_i \leq^{\mathsf{Tm}} t, y \; R \; t_i, \Delta' \vdash p \ \text{for } i = 1, \ldots, n}{\Delta, y \; R \; \mathrm{case}(t, \boldsymbol{x}.\boldsymbol{t}), \Delta' \vdash p} \qquad (R\mathrm{piL})\ \frac{\Delta, y \; R \; t, \Delta' \vdash p}{\Delta, y \; R \; \mathrm{pi}_j(t), \Delta' \vdash p}$$

$$(R\mathrm{appL})\ \frac{\Delta, y \; R \; s, \Delta' \vdash p}{\Delta, y \; R \; s \; a, \Delta' \vdash p} \qquad (R\mathrm{unfL})\ \frac{\Delta, y <^{\mathsf{Tm}} t, \Delta' \vdash p}{\Delta, y \; R \; \mathrm{unfold}(t), \Delta' \vdash p}$$

Reflexivity and transitivity:

$$(\leq^{\mathsf{Tm}}\mathrm{refl})\ \frac{}{\Delta \vdash t \leq^{\mathsf{Tm}} t}$$

$$(<^{\mathsf{Tm}}\mathrm{transL})\ \frac{\Delta \vdash s \; R \; t \qquad y <^{\mathsf{Tm}} s \in \Delta \qquad R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}}{\Delta \vdash y <^{\mathsf{Tm}} t}$$

$$(<^{\mathsf{Tm}}\mathrm{transR})\ \frac{\Delta \vdash s <^{\mathsf{Tm}} t \qquad y \; R \; s \in \Delta \qquad R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}}{\Delta \vdash y <^{\mathsf{Tm}} t}$$

$$(\leq^{\mathsf{Tm}}\mathrm{trans})\ \frac{\Delta \vdash s \; R \; t \qquad y \; S \; s \in \Delta \qquad R, S \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}}{\Delta \vdash y \leq^{\mathsf{Tm}} t}$$

Table 5. Structural ordering on Terms

Declaration 1 (Structural ordering). *The structural ordering on terms $<^{\mathsf{Tm}}$ and its non-strict version $\leq^{\mathsf{Tm}}$ are defined as families of relations indexed over a pair of types: For all $\sigma, \tau \in \mathsf{Ty}$ we define*

$$<^{\mathsf{Tm}}_{\sigma,\tau} \subseteq \mathsf{Tm}^{\sigma} \times \mathsf{Tm}^{\tau}$$
$$\leq^{\mathsf{Tm}}_{\sigma,\tau} \subseteq \mathsf{Tm}^{\sigma} \times \mathsf{Tm}^{\tau}$$

For purposes of readability we will generally omit the indices.

Definition 4 (Dependencies). *A set of dependencies Δ consists of relations*

$$y \; R \; t \qquad \textit{where } y \in \mathsf{TmVar}^{\sigma},\; t \in \mathsf{Tm}^{\tau},\; R \in \{<^{\mathsf{Tm}}_{\sigma,\tau}, \leq^{\mathsf{Tm}}_{\sigma,\tau}\}$$

Definition 5 (Derivation of structural ordering). *By the rules in Table 5 we introduce the judgement*

$$\Delta \vdash s \; R \; t \qquad R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}$$

Read "under the dependencies Δ we know that s is less (or equal) than t".

The right hand rules work on the judgements we want to derive, whereas the left hand side rules work on the dependencies, which—in backward reading—are only introduced by the treatment of case statements (rules (RcaseR) and (srcase)—see next section). Therefore the dependencies can be restricted to the form $y \; R^{\mathsf{Tm}} \; t$, where y is the fresh variable introduced for one clause in the case statement. Typically t, the term that is analyzed by the case expression, will be of the form $\mathrm{unfold}(t')$, hence of the left-rules ($<^{\mathsf{Tm}}$unfL) will be the one mostly used in practice. To see the system "in action", we give derivations of the proof obligations for **add** (omitting the superindex $^{\mathsf{Tm}}$):

$$\dfrac{\dfrac{\dfrac{}{n < y \vdash y \leq y}\leq^{\mathsf{Tm}}\mathrm{refl}}{n < y \vdash n < y}<^{\mathsf{Tm}}\mathrm{transL}}{n \leq \mathrm{unfold}(y) \vdash n < y}\leq^{\mathsf{Tm}}\mathrm{unfL} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{f < y \vdash y \leq y}\leq^{\mathsf{Tm}}\mathrm{refl}}{f < y \vdash f < y}<^{\mathsf{Tm}}\mathrm{transL}}{f \leq \mathrm{unfold}(y) \vdash f < y}\leq^{\mathsf{Tm}}\mathrm{unfL}}{f \leq \mathrm{unfold}(y) \vdash f\; z < y}<^{\mathsf{Tm}}\mathrm{appR}$$

The rule ($<^{\mathsf{Tm}}$caseL) is needed for nested case statements, as for instance in the following curious implementation of the "half"-function:

$$\begin{aligned}\mathrm{fun}\;\mathrm{half}^{\mathbf{Nat}\to\mathbf{Nat}}(n^{\mathbf{Nat}}) = \mathrm{case}(&\mathrm{unfold}(\mathrm{case}(\mathrm{unfold}(n),\\ &\qquad\qquad\quad \_^{1}.\, n,\\ &\qquad\qquad\quad n_1^{\mathbf{Nat}}.\, n_1)),\\ &\_^{1}.\, \mathbf{O},\\ &n_2^{\mathbf{Nat}}.\, \mathbf{S}(\mathrm{half}(n_2)))\end{aligned}$$

The obligation $n_2 <^{\mathsf{Tm}} n$ is proven as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{\_ \leq \mathrm{unfold}(n), n_2 < n \vdash n_2 < n} \quad \dfrac{\dots}{n_1 \leq \mathrm{unfold}(n), n_2 < n_1 \vdash n_2 < n}}{n_2 < \mathrm{case}(\mathrm{unfold}(n), \_.n, n_1.n_1) \vdash n_2 < n}<^{\mathsf{Tm}}\mathrm{caseL}}{n_2 \leq \mathrm{unfold}(\mathrm{case}(\mathrm{unfold}(n), \_.n, n_1.n_1)) \vdash n_2 < n}\leq^{\mathsf{Tm}}\mathrm{unfL}$$

3.2 Lexicographic Extension

To handle functions like the Ackermann function, we need extend our calculus to lexicographic orderings. This requires just two additional rules.

Declaration 2 (Lexicographic ordering). *Given closed types* $\boldsymbol{\sigma} = \sigma_1, \ldots, \sigma_n$ *and a permutation* $\pi \in S_n$ *we define the relation*

$$(s_1, ..., s_n) \prec^{\mathsf{Tm}}_{\pi,\boldsymbol{\sigma}} t \qquad \textit{where } s_1 \in \mathsf{Tm}^{\sigma_1}, \ \ldots, s_n \in \mathsf{Tm}^{\sigma_n}, \ t \in \mathsf{Tm}^{\Pi\boldsymbol{\sigma}}$$

To enhance readability we will usually omit the second index $\boldsymbol{\sigma}$*.*

By this definition we mean that term $(\boldsymbol{s})$ is lexicographically smaller than term t w.r.t. a permutation π of the components. Note that the left hand side must be a tuple syntactically, whereas the right hand side may be any term of product type.

Definition 6 (Derivation of lexicographic ordering). *By the following rules we introduce an auxiliary judgement*

$$\Delta \vdash^k (\boldsymbol{s}) \prec^{\mathsf{Tm}}_{\pi} t \qquad 1 \leq k \leq n = |\boldsymbol{s}|$$

In case of $k = 1$ *we just write*

$$\Delta \vdash (\boldsymbol{s}) \prec^{\mathsf{Tm}}_{\pi} t$$

Read "under the dependencies Δ *we know that* $(\boldsymbol{s})$ *is lexicographically smaller than* t *w.r.t. the permutation* π*".*

$$(\mathrm{lex}{<}^{\mathsf{Tm}})\ \frac{\Delta \vdash s_{\pi(k)} <^{\mathsf{Tm}} \mathrm{pi}_{\pi(k)}(t)}{\Delta \vdash^k (\boldsymbol{s}) \prec^{\mathsf{Tm}}_{\pi} t}$$

$$(\mathrm{lex}{\leq}^{\mathsf{Tm}})\ \frac{\Delta \vdash s_{\pi(k)} \leq^{\mathsf{Tm}} \mathrm{pi}_{\pi(k)}(t) \qquad \Delta \vdash^{k+1} (\boldsymbol{s}) \prec^{\mathsf{Tm}}_{\pi} t}{\Delta \vdash^k (\boldsymbol{s}) \prec^{\mathsf{Tm}}_{\pi} t}$$

This encodes the standard lexicographic ordering. We start in comparing the first component ($k = 1$) of tuple $(\boldsymbol{s})$ with tuple t. If it is only non-strictly smaller, we have to consider the next component ($k \leadsto k+1$). The terms "first" and "next" have to be seen relatively to the permuation π.

3.3 Structural Recursiveness

As a frame for the derivation system for size relations on terms, we now define the judgement "sr" that we introduced in Sect. 2. Roughly described, a function g will be structurally recursive in a term t, if it is so in all subterms of t and is called recursively only with smaller arguments (see rule (srapprec)). This is were a reference to the judgement "$\prec^{\mathsf{Tm}}$" is made.

Definition 7 (Derivation of structural recursiveness). *We introduce the judgement*

$$\Delta \vdash g(x) \ \mathsf{sr}_{\pi} \ t \qquad \textit{where } g \in \mathsf{TmVar}^{\sigma \to \tau}, \ x \in \mathsf{TmVar}^{\sigma}, \ t \in \mathsf{Tm}^{\tau},$$

read "under the dependencies Δ the function g with parameter x is structurally recursive in t w.r.t. the permutation π", by the following rules.

$$(\text{srvar})\ \frac{y \neq g}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ y} \qquad (\text{srweak})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t[\Gamma]}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t[\Gamma, x]} \qquad (\text{srin})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ \text{in}_j(t)}$$

$$(\text{srcase})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ s \qquad \Delta, x_i \leq^{\mathsf{Tm}} s \vdash g(x)\ \mathsf{sr}_\pi\ t_i \ \text{ for } i = 1, ..., |\boldsymbol{t}|}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ \text{case}(s, \boldsymbol{x.t})}$$

$$(\text{srtup})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t_i \ \text{ for } i = 1, ..., |\boldsymbol{t}|}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ (\boldsymbol{t})} \qquad (\text{srpi})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ \text{pi}_j(t)}$$

$$(\text{srlam})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t \qquad y \notin \{g, x\}}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ \lambda y.\, t} \qquad (\text{srapp})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t \qquad \Delta \vdash g(x)\ \mathsf{sr}_\pi\ s}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t\, s}$$

$$(\text{srapprec})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ (\boldsymbol{a}) \qquad \Delta \vdash (\boldsymbol{a}) \prec_\pi^{\mathsf{Tm}} x}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ g(\boldsymbol{a})}$$

$$(\text{srfold})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ \text{fold}(t)} \qquad (\text{srunfold})\ \frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t}{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ \text{unfold}(t)}$$

Note that g, x and π remain fixed in all rules. Furthermore, since there is no rule for recursive terms and since "sr" is used in the term definition, in our system a nested definition of functions, and thus mutual recursion, is not possible.

Definition 8 (Syntactically structurally recursive). *We define a recursive term* $\text{fun}\, g(x) = t$ *to be syntactically structurally recursive,*

$$\Delta \vdash g(x)\ \mathsf{sr}\ t$$

if there is a permutation π s.th. $\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t$.

As an example we show that **add** is a definable term in the foetus system. Expanding the syntactic sugar and the abbreviations and omitting some type annotations the term becomes

$$\begin{aligned} \lambda x.\ \text{fun add}(\bar{y}^{\Pi(\mathbf{Ord})}) = \text{case}(&\text{unfold}(\text{pi}_1(\bar{y})),\\ &\_.\ x,\\ &n.\ \text{fold}(\text{in}_2(\text{add}(n))),\\ &f.\ \text{fold}(\text{in}_3(\lambda z.\ \text{add}(f\, z)))) \end{aligned}$$

To prove $\mathbf{add} \in \mathsf{Tm}^{\mathbf{Ord} \to \Pi(\mathbf{Ord}) \to \mathbf{Ord}}$ we have to show that

$$\vdash \text{add}(\bar{y})\ \mathsf{sr}_\pi\ \text{case}(\dots)$$

where π is the identical permutation on S_1. We infer our goal by (srcase), obtaining four subgoals:

– Head term:

$$\dfrac{\dfrac{\dfrac{\bar{y} \neq \mathrm{add}}{\vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \bar{y}}\,\mathrm{srvar}}{\vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{pi}_1(\bar{y})}\,\mathrm{srpi}}{\vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{unfold}(\mathrm{pi}_1(\bar{y}))}\,\mathrm{srunfold}$$

– Side term 1:

$$\dfrac{x \neq \mathrm{add}}{\_ \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ x}\,\mathrm{srvar}$$

– Side terms 2 and 3: We prove them by reusing the derivations for structural ordering in Sect. 3.1, substituting $\mathrm{pi}_1(\bar{y})$ for y in all occurrences.

$$\dfrac{\dfrac{\dfrac{\dfrac{n \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash n <^{\mathsf{Tm}} \mathrm{pi}_1(\bar{y})}{n \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash (n) \prec_\pi^{\mathsf{Tm}} \bar{y}}\,\mathrm{lex}{<^{\mathsf{Tm}}}}{n \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{add}(n)}\,\mathrm{srapprec}}{n \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{in}_2(\mathrm{add}(n))}\,\mathrm{srin}}{n \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{fold}(\mathrm{in}_2(\mathrm{add}(n)))}\,\mathrm{srfold}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{z \notin \{\mathrm{add}, \bar{y}\} \quad \dfrac{f \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash f\ z <^{\mathsf{Tm}} \mathrm{pi}_1(\bar{y})}{f \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash (f\ z) \prec_\pi^{\mathsf{Tm}} \bar{y}}\,\mathrm{lex}{<^{\mathsf{Tm}}}}{z \notin \{\mathrm{add}, \bar{y}\} \quad f \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{add}(f\ z)}\,\mathrm{srapprec}}{f \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \lambda\mathbf{z}.\,\mathrm{add}(f\ z)}\,\mathrm{srlam}}{f \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{in}_3(\lambda z.\,\mathrm{add}(f\ z))}\,\mathrm{srin}}{f \leq^{\mathsf{Tm}} \mathrm{unfold}(\mathrm{pi}_1(\bar{y})) \vdash \mathrm{add}(\bar{y})\ \mathsf{sr}_\pi\ \mathbf{fold}(\mathrm{in}_3(\lambda z.\,\mathrm{add}(f\ z)))}\,\mathrm{srfold}$$

4 Soundness of the Structural Ordering

In this section we show that the ordering on values corresponds to the structural ordering on terms. We accomplish this by proving that evaluation preserves the structural ordering. To this end, we give an interpretation of the judgement $\Delta \vdash s\ R^{\mathsf{Tm}}\ t$:

Definition 9 (Weak and strong interpretation of the structural ordering). *We define the propositions "environment e satisfies (weakly) the relation $s\ R^{\mathsf{Tm}}\ t$ for the terms s and t" and "e satisfies the dependencies Δ":*

$$\begin{array}{lll} e \vDash^{\mathsf{wk}} s\ R^{\mathsf{Tm}}\ t & :\Longleftrightarrow & \forall v, w.\ \langle s; e\rangle \Downarrow v \rightarrow \langle t; e\rangle \Downarrow w \rightarrow v\ R\ w \\ e \vDash s\ R^{\mathsf{Tm}}\ t & :\Longleftrightarrow & \exists v, w.\ \langle s; e\rangle \Downarrow v\ \&\ \langle t; e\rangle \Downarrow w\ \&\ v\ R\ w \\ e \vDash \Delta & :\Longleftrightarrow & \forall p \in \Delta.\ e \vDash p \end{array}$$

Strong satisfaction of an atom p of the form $s\ R^{\mathsf{Tm}}\ t$ carries evidence that s and t are strongly evaluating, whereas weak satisfaction needs proof of this. We will interpret atoms in the dependencies (left hand side) strongly and the concluded atom (right hand side) weakly. The reason for this assymetry lies in the architecture the sr-judgement. Interpreting terms t for which the judgment $\Delta \vdash g(x)\ \mathsf{sr}\ t$ holds as strongly terminating—which we will do in Sect. 5—we can read off the definition of sr that only strongly evaluating terms enter the dependencies.

Lemma 1 (Weakening). *Extending the environment does not destroy satisfaction of dependencies.*

$$\frac{e \vDash^{\mathsf{wk}} p}{e, x = v \vDash p} \qquad \frac{e \vDash \Delta}{e, x = v \vDash \Delta}$$

Proof. Since by the definition of contexts and environments x must be a new variable, it does not appear in e and thus not in any of the terms in p or Δ.

Theorem 1. *The structural ordering on terms and its lexicographic extension are preserved by the operational semantics.*

$$\frac{\Delta \vdash p}{\forall e \vDash \Delta.\ e \vDash^{\mathsf{wk}} p}$$

Proof. By induction on $\Delta \vdash p$.
Right hand side rules ($R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}$):

(RcaseR) We have to show

$$\frac{\forall i, e' \vDash (\Delta, x_i \leq^{\mathsf{Tm}} s).\ e' \vDash^{\mathsf{wk}} s_i\ R^{\mathsf{Tm}}\ t \quad e \vDash \Delta \quad \langle \mathrm{case}(s, \boldsymbol{x.s}); e\rangle \Downarrow v \quad \langle t; e\rangle \Downarrow w}{v\ R\ w}$$

The assumption $\langle \mathrm{case}(s, \boldsymbol{x.s}); e\rangle \Downarrow$ entails by (opcase^{-1}) $\langle s; e\rangle \downarrow \mathrm{in}_j(v')$, and since s is a subterm of $\mathrm{case}(s, \boldsymbol{x.s})$, by definition 3 also $\mathrm{in}_j(v') \in \mathsf{VAL}$. By an instance of the induction hypothesis using j and $e' \equiv (e, x_j = v')$ (which is of course a good environment) we obtain $e' \vDash^{\mathsf{wk}} s_j\ R^{\mathsf{Tm}}\ t$ (*) leaving us four subgoals.
1. $e' \vDash \Delta$: by weakening (lemma 1)
2. $\langle x_j; e'\rangle \Downarrow v'$: by (opvar)
3. $\langle s; e'\rangle \Downarrow \mathrm{in}_j(v')$: by (opweak)
4. $v' \leq \mathrm{in}_j(v')$: by ($\leq$in)

Since by (opcase^{-1}) also $\langle s_j; e'\rangle \Downarrow v$ and by (opweak) $\langle t; e'\rangle \Downarrow w$ we can infer our goal $v\ R\ w$ from (*).

(RpiR) Our goal is

$$\frac{\langle s; e\rangle \Downarrow (\boldsymbol{v}) \to \langle t; e\rangle \Downarrow w \to (\boldsymbol{v})\ R\ w \qquad \langle \mathrm{pi}_j(s); e\rangle \Downarrow v_j \qquad \langle t; e\rangle \Downarrow w}{v_j\ R\ w}$$

Since (oppi^{-1}) entails $\langle s; e\rangle \Downarrow (\boldsymbol{v})$ because s is a subterm of $\mathrm{pi}_j(s)$, we can use the ind.hyp. and achieve our goal using (Rtup').

(RappR) Here we show

$$\frac{\langle s; e\rangle \Downarrow f \to \langle t; e\rangle \Downarrow w \to f\ R\ w \qquad \langle sa; e\rangle \Downarrow v \qquad \langle t; e\rangle \Downarrow w}{v\ R\ w}$$

By (opapp^{-1}) and the subterm property $\langle s; e\rangle \Downarrow f$, $\langle a; e\rangle \Downarrow u$ and $f@u \Downarrow v$. Hence we complete using the ind.hyp. and the rule (Rarr').

(RunfR) analogously using ($\mathrm{opunfold}^{-1}$) and (Rfold')

Left hand side rules ($R \in \{<^{\mathsf{Tm}}, \leq^{\mathsf{Tm}}\}$): All the goals we have to show are of the form

$$\frac{\forall e' \vDash (\Delta, q', \Delta').\ e' \vDash^{\mathsf{wk}} p}{\forall e \vDash (\Delta, q, \Delta').\ e \vDash^{\mathsf{wk}} p}$$

Hence by weakening it suffices to show $e' \vDash q'$ from $e \vDash q$ for each case, where e' is e or an extension of e.

(RcaseL) Assume $e \vDash y\ R\ \mathrm{case}(t, \boldsymbol{x}.\boldsymbol{t})$, which expands to the three propositions $\langle y; e\rangle \Downarrow v$ (1), $\langle \mathrm{case}(t, \boldsymbol{x}.\boldsymbol{t}); e\rangle \Downarrow w$ (2) and $v\ R\ w$ (3). The rule (opcase^{-1}) plus subterm property entails $\langle t; e\rangle \Downarrow \mathrm{in}_j(v')$ (2a) and $\langle t_j; e, x_j = v'\rangle \Downarrow w$ (2b). Our two goals are:

1. $e, x_j = v' \vDash x_j \leq^{\mathsf{Tm}} t$: We prove this by $\langle x_j; e, x_j = v'\rangle \Downarrow v'$ (opvar), by (2a) and by $v' \leq \mathrm{in}_j(v')$ ($\leq$in).
2. $e, x_j = v' \vDash y\ R\ t_j$: By (1), (2b) and (3) using weakening.

(RpiL) We expand our goal to

$$\frac{\langle y; e\rangle \Downarrow v \qquad \langle \mathrm{pi}_j(t); e\rangle \Downarrow w_j \qquad v\ R\ w_j}{\langle y; e\rangle \Downarrow v \qquad \langle t; e\rangle \Downarrow (\boldsymbol{w}) \qquad v\ R\ (\boldsymbol{w})}$$

It follows from (oppi^{-1}), subterm property and (Rtup).

(RappL) The expanded goal is

$$\frac{\langle y; e\rangle \Downarrow v \qquad \langle s\, a; e\rangle \Downarrow w \qquad v\ R\ w}{\langle y; e\rangle \Downarrow v \qquad \langle s; e\rangle \Downarrow f \qquad v\ R\ f}$$

By (opapp^{-1}) and subterm property $\langle s; e\rangle \Downarrow f$, $\langle a; e\rangle \Downarrow u$ and $f@u \Downarrow w$. Thus $v\ R\ f$ follows by (Rarr).

(RunfL) analogously using ($\mathrm{opunfold}^{-1}$) and (Rfold)

Reflexivity and transitivity:

($\leq^{\mathsf{Tm}}$refl) $e \vDash^{\mathsf{wk}} t \leq^{\mathsf{Tm}} t$ follows from ($\leq$refl).

($<^{\mathsf{Tm}}$transL) We have to show ($R \in \{<, \leq\}$)

$$\frac{e \vDash^{\mathsf{wk}} s \; R^{\mathsf{Tm}} \; t \qquad e \vDash y <^{\mathsf{Tm}} s \qquad \langle y; e\rangle \Downarrow u \qquad \langle t; e\rangle \Downarrow w}{u < w}$$

From $e \vDash y <^{\mathsf{Tm}} s$ we obtain $\langle s; e\rangle \Downarrow v$ and $u < v$ and thus by the first premise $v \; R \; w$. Transitivity of the structural ordering on values implies $u < w$.

($<^{\mathsf{Tm}}$transR) analogously

($\leq^{\mathsf{Tm}}$trans) analogously

Lexicographic extension:

(lex$<^{\mathsf{Tm}}$) We have the simplified goal

$$\frac{e \vDash^{\mathsf{wk}} s_{\pi(k)} <^{\mathsf{Tm}} \mathrm{pi}_{\pi(k)}(t) \qquad \langle(\boldsymbol{s}); e\rangle \Downarrow (\boldsymbol{v}) \qquad \langle t; e\rangle \Downarrow (\boldsymbol{w})}{(\boldsymbol{v}) \prec_{\pi}^{k} (\boldsymbol{w})}$$

By (optup^{-1}) $\langle s_{\pi(k)}; e\rangle \Downarrow v_{\pi(k)}$ and by (oppi) $\langle \mathrm{pi}_{\pi(k)}(t); e\rangle \Downarrow w_{\pi(k)}$ hence $v_{\pi(k)} < w_{\pi(k)}$, and the goal $(\boldsymbol{v}) \prec_{\pi}^{k} (\boldsymbol{w})$ follows form (lex<).

(lex$\leq^{\mathsf{Tm}}$) Our simplified goal is

$$\frac{e \vDash^{\mathsf{wk}} s_{\pi(k)} \leq^{\mathsf{Tm}} \mathrm{pi}_{\pi(k)}(t) \qquad (\boldsymbol{v}) \prec_{\pi}^{k+1} (\boldsymbol{w}) \qquad \langle(\boldsymbol{s}); e\rangle \Downarrow (\boldsymbol{v}) \qquad \langle t; e\rangle \Downarrow (\boldsymbol{w})}{(\boldsymbol{v}) \prec_{\pi}^{k} (\boldsymbol{w})}$$

It follows analogously to (lex$<^{\mathsf{Tm}}$) using (lex$\leq$). □

5 Soundness of Structural Recursion

We transfer the syntactic property of being structurally recursive to our semantics. Then we show that every structurally recursive term is good.

Definition 10 (Semantically structurally recursive). *We say a function value $f \in \mathsf{Val}^{\sigma\to\tau}$ is semantically structurally recursive $f \in \mathsf{SR}^{\sigma\to\tau}$ if it terminates on all inputs v under the condition that it terminates on all lexicographically smaller inputs $w \prec v$:*

$$f \in \mathsf{SR}^{\sigma\to\tau} :\Longleftrightarrow \exists\pi\forall v \in \mathsf{VAL}^{\sigma}.\ (\forall w \in \mathsf{VAL}^{\sigma}.\ w \prec_{\pi} v \to f@w \Downarrow) \to f@v \Downarrow$$

Proposition 1. $\mathsf{SR}^{\sigma\to\tau} = \mathsf{VAL}^{\sigma\to\tau}$

Proof. The domain VAL^{σ} of all function values of type $\sigma \to \tau$ is wellfounded w.r.t. the lexicographic ordering, what we have shown in detail in [Abe99] and [AA99]. Thus the wellfounded induction principle establishes the equality between semantically structurally recursive and good functions. □

Theorem 2. *Every recursive term of type $\sigma \to \tau$ and context Γ is good in a good initial environment $e_0 \in \mathsf{VAL}(\Gamma)$.*

$$\langle \text{fun}\, g(x) = t_0; e_0 \rangle \in \mathsf{VAL}^{\sigma \to \tau}$$

Proof. By definition there exists a permutation π s.th. $\vdash g(x)\ \mathsf{sr}_\pi\ t_0$. Using the abbreviation

$$f_0 \equiv \langle \text{fun}\, g(x) = t_0; e_0 \rangle$$

and proposition 1 our goal becomes $(\forall w \in \mathsf{VAL}^\sigma.\ w \prec v_0 \to f_0 @ w \Downarrow) \to f_0 @ v_0 \Downarrow$. If we can prove the following lemma under the global assumption

$$\forall w \prec v_0.\ f_0 @ w \Downarrow \tag{1}$$

we can finish using this lemma with empty Δ, $t \equiv t_0$ and $e \equiv (e_0, g = f_0, x = v_0)$.

Lemma 2. $\dfrac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t \qquad e \vDash \Delta \qquad \langle g; e \rangle \downarrow f_0 \qquad \langle x; e \rangle \Downarrow v_0}{\langle t; e \rangle \Downarrow}$

Proof. By induction on $\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t$.

(srvar) $\langle y; e \rangle \Downarrow$ since e is good except for g, but $y \neq g$ by assumption.

(srcase) We have to show

$$\frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ s \quad \forall i.\ \Delta, x_i \leq^{\mathsf{Tm}} s \vdash g(x)\ \mathsf{sr}_\pi\ t_i \quad e \vDash \Delta \quad \langle g; e \rangle \downarrow f_0 \quad \langle x; e \rangle \Downarrow v_0}{\langle \text{case}(s, \boldsymbol{x.t}); e \rangle \Downarrow}$$

The first ind.hyp. entails $\langle s; e \rangle \Downarrow \text{in}_j(v')$. By (opcase) our goal follows from the second ind.hyp. using environment $e' \equiv e, x_j = v'$, if we show the three premises of the ind.hyp.:

1. $e' \vDash \Delta, x_j \leq^{\mathsf{Tm}} s$: This follows from weakening and the three facts $\langle x_j; e' \rangle \Downarrow v'$, $\langle s; e' \rangle \Downarrow \text{in}_j(v')$ and $v' \leq \text{in}_j(v')$.
2. $\langle g; e' \rangle \downarrow f_0$: by (opweak)
3. $\langle x; e' \rangle \Downarrow v_0$: by (opweak)

(srlam) We have to show

$$\frac{\Delta \vdash g(x)\ \mathsf{sr}_\pi\ t \qquad e \vDash \Delta \qquad \langle g; e \rangle \downarrow f_0 \qquad \langle x; e \rangle \Downarrow v_0}{\langle \lambda y.t; e \rangle \Downarrow}$$

Immediately by (oplam) we get $\langle \lambda y.t; e \rangle \downarrow$, thus it remains to show $\langle \lambda y.\, t; e \rangle \in \mathsf{VAL}$. For this we assume $u \in \mathsf{VAL}$ and show $\langle \lambda y.t; e \rangle @ u \Downarrow$. The latter follows from (opappvl) and the ind.hyp. $\langle t; e' \rangle \Downarrow$ for $e' \equiv (e, y = u)$, since $e' \vDash \Delta$ by weakening and $\langle g; e' \rangle \downarrow f_0$ and $\langle x; e' \rangle \Downarrow v_0$ by (opweak).

(srapprec) Using the ind.hyp. our goal becomes

$$\frac{\langle(\boldsymbol{a}); e\rangle \Downarrow w \qquad \Delta \vdash (\boldsymbol{a}) \prec_{\pi}^{\mathsf{Tm}} x \qquad e \vDash \Delta \quad \langle g; e\rangle \downarrow f_0 \quad \langle x; e\rangle \Downarrow v_0 \quad w \prec_{\pi} v_0}{\langle g(\boldsymbol{a}); e\rangle \Downarrow}$$

By (opapp) and (opappvr) this is true, if $f_0@w \Downarrow$. That, however, was our global assumption (1) for $w \prec_{\pi} v_0$, which we obtain if we apply theorem 1 on the premise $\Delta \vdash (\boldsymbol{a}) \prec_{\pi}^{\mathsf{Tm}} x$.

All other cases follow directly by ind.hyp. using the operational semantics. □

Corollary 1. *All terms t terminate in a good environment e.*

$$\forall t^{\sigma}[\Gamma], e \in \mathsf{ENV}[\Gamma].\ \exists v \in \mathsf{VAL}^{\sigma}.\ \langle t; e\rangle \Downarrow v$$

Proof. By straightforward induction on t, using the operational semantics. For the critical case $t \equiv \mathrm{fun}\, g(x) = s$ use theorem 2. The proof has been carried out in [Abe99] and [AA99].

Corollary 2. *All closed terms terminate.*

From this corollary we can extract an interpreter for the foetus language that always terminates. This is no surprise since the interpreter just applies the operational semantics on the input term. Additionally, it computes a witness for the goodness of the result value ($v \in \mathsf{VAL}$), which could be eliminated, using a refined program extraction (cf. [BS93]).

6 Conclusions and Further Work

We have formally defined a syntactical check "sr" for structurally recursive functions that serves as a frame for the derivation system for size relations between terms given in [AA99]. We have shown that these two parts of the termination checker are sound w.r.t. our operational semantics.

I expect that my approach can be extended to mutual recursion (see below) and dependent types, since they only put more restrictions on the acceptable terms. By this I mean that every term typable in a lambda calculus with inductive and dependent types ($\lambda\Pi$) should be typable in foetus. Hence we could just strip the dependency and run the foetus termination checker. My standpoint is confirmed by the fact that implementations of a termination checker for $\lambda\Pi$ do not make use of the typing information (cf. [PP00]).

So far the termination checking of foetus is very limited, e.g., "quicksort" cannot be proven total with our method. To capture the Walther recursive functions [MA96] like quicksort one has to define two more judgements stating that a function is *reducing* resp. *preserving*. E.g., for quicksort the filtering step has to be preserving. Implementation of this so-called *reduction checking* should be straightforward for the simple structural ordering on terms. However, lexicographic orderings will require a number of modifications since they are not "first class citizens" in my system so far. They may appear only on right hand sides, not within the dependencies.

In [Abe98] and [AA99] Altenkirch and myself have informally described a termination checker also for mutual recursive functions. The main extension is the construction of a call graph for the mutual recursive functions, which has to satisfy a "goodness" condition. This enables the construction of a wellfounded ordering on the function symbols which, in addition to the lexicographic ordering on the arguments, serves as a component of the termination ordering required to run through the soundness proof. Work on the details is in progress.

In contrast to the full approach with call graphs a light weight version of mutual recursion with descent in every call would be a straightforward extension of the proof in the present article.

6.1 Acknowledgements

A number of colleagues deserve my thanks for their contribution to this work: Thorsten Altenkirch, for laying the foundations and for discussions on the definitions of the judgements and their semantics; Brigitte Pientka, for discussions on the commonesses and differences of our formal systems; and Ralph Matthes, Aaron Stump and the two anonymous referees for helpful comments on the draft.

References

[AA99] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. Submitted to the Journal of Functional Programming, December 1999.

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

[Abe99] Andreas Abel. A semantic analysis of structural recursion. Master's thesis, Ludwig-Maximilians-University Munich, 1999. http://www.informatik.uni-muenchen.de/~abel/publications/.

[BJO00] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive data type systems. To appear in Theoretical Computer Science, 2000.

[BS93] Ulrich Berger and Helmut Schwichtenberg. Program Development by Proof Transformation. Talk at Marktoberdorf, draft available under http://www.mathematik.uni-muenchen.de/~schwicht/, 1993.

[Coq92] Thierry Coquand. Pattern matching with dependent types. WWW, 1992.

[Coq99] Catarina Coquand. Agda home page. http://www.cs.chalmers.se/~catarina/agda/, 1999.

[Gim95] Eduardo Gimenéz. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Bøastad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *LNCS*, pages 39–59. Springer, 1995.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association of Computing Machinery*, 40(1):143–184, January 1993.

[JO97] J. P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173, 1997.

[MA96] David McAllester and Kostas Arkoudas. Walther Recursion. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 – August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[PP00] Brigitte Pientka and Frank Pfenning. Termination and Reduction Checking in the Logical Framework. *Workshop on Automation of Proofs by Mathematical Induction*, CADE-17, Pittsburgh, PA, USA, June 2000.

[PS98] Frank Pfenning and Carsten Schürmann. Twelf user's guide. Technical report, Carnegie Mellon University, 1998.

[TT99] A.J. Telford and D.A. Turner. Ensuring Termination in ESFP. Submitted to BCTCS 15 proceedings issue of Journal of Universal Computer Science, 1999.