A Predicative Analysis of Structural Recursion

ANDREAS ABEL*
Department of Computer Science, University of Munich, 80538 Munich, Germany
(*e-mail:* abel@tcs.informatik.uni-muenchen.de)

THORSTEN ALTENKIRCH
School of Computer Science & Information Technology, University of Nottingham, Nottingham, NG8 1BB, UK
(*e-mail:* txa@cs.nott.ac.uk)

Abstract

We introduce a language based upon lambda calculus with products, coproducts and strictly positive inductive types that allows the definition of recursive terms. We present the implementation (foetus) of a syntactical check that ensures that all such terms are structurally recursive, i.e., recursive calls appear only with arguments structurally smaller than the input parameters of terms considered. To ensure the correctness of the termination checker, we show that all structurally recursive terms are normalizing with respect to a given operational semantics. To this end, we define a semantics on all types and a structural ordering on the values in this semantics and prove that all values are accessible with regard to this ordering. Finally, we point out how to do this proof predicatively using set based operators.

1 Introduction

In lambda calculi with inductive types the standard means to construct a function over an inductive type is the *recursor*, which corresponds to *induction*. This method, however, has several drawbacks, as discussed in (Coquand, 1992). One of them is that programs are hard to understand intuitively. E.g., the "division by 2"-function may be coded with recursors over natural numbers R^N and booleans R^B as follows:

$$\begin{array}{lcl} R^N & : & \sigma \to (N \to \sigma \to \sigma) \to N \to \sigma \\ R^B & : & \sigma \to \sigma \to B \to \sigma \\ \mathit{half} & = & \lambda n^N .\, R^N\, (\lambda x^B .\, 0)\, (\lambda x^N \lambda f^{B \to N} .\, R^B\, (f\ \mathit{true})\, (1 + (f\ \mathit{false})))\ n\ \mathit{false} \end{array}$$

Alternatively, in the presence of products, it can be implemented involving an auxiliary function returning a pair. But in both cases additional constructs are unavoidable. Therefore, a concept parallel to *complete* resp. *wellfounded* induction has been investigated: *recursive definitions with pattern matching*, as they are common in most functional programming languages. In SML our example could be written down straightforward as follows:

* Current address: CMU Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA.
This work was supported by the Graduiertenkolleg Logik in der Informatik (DFG)

```
datatype Nat = O | S of Nat

fun half O         = O
  | half (S O)     = O
  | half (S (S n)) = S (half n)
```

These recursive definitions with pattern matching have to satisfy two conditions to define total functions:

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 totality of the pattern is always given. For a discussion see (Coquand, 1992).
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 the arguments are smaller than the input parameters in each recursive call of that function.

How to find such termination orderings has been dealt with comprehensively in the area of termination of rewriting (Dershowitz, 1987) and functional programs (Giesl, 1995). We restrict to *structural* orderings and their lexicographic extensions since they are sufficient in many cases and our focus is on a theoretical foundation rather than on completeness.

1.1 Structural Ordering

Usually a value v' is considered *structurally smaller* than another value v if $v < v'$ w.r.t. to the component resp. subterm ordering. It can be defined as the transitive closure of

$$w < \mathrm{C}(\ldots, w, \ldots) \tag{1}$$

where C is a constructor of an inductive datatype. On this basis a structural ordering $t < t'$ on terms t, t' can be defined to hold iff $v < v'$ where t evaluates to v and t' to v'. An alternative, which we pursue in this paper, is to define a structural ordering on terms independently and show that evaluation maps it into the ordering on values.

The axiom (1) is motivated by term structure. We will consider another axiom, motivated by the *type* structure:

$$(f\,a)^{\tau} \leq f^{\sigma \to \tau} \tag{2}$$

An informal justification for this axiom can be drawn from set theory where a function is defined as the set of all argument-result-pairs. Thus *one* result $f\,a$ can be seen as a component of f which contains *all* possible results.

To verify in detail that the order induced by (1) and (2) is wellfounded is a major contribution of this article. It relies crucially on the *predicativity* of the type system. In our case this means that all inductive types are strictly positive. Furthermore we exclude impredicative polymorphism which destroys the wellfoundedness of the structural ordering as exemplified by Coquand (1992).

Consider the following program written in Haskell with 2nd order polymorphism[1]:

```
data V = C (forall a.a -> a)

f :: V -> V
f (C x) = f (x (C x))
```

The argument of the constructor `C` is a polymorphic function of type $\forall a.\, a \to a$, like `id`. Since

$$\texttt{C id} > \texttt{id} \geq \texttt{id (C id)} = \texttt{C id}$$

the structurual ordering is no longer wellfounded, and the function `f`, applied to `C id`, loops infinitely.

1.2 Structural Recursion

We define a *structurally recursive* function as

> a recursively defined function which calls itself (directly or indirectly) only for structurally smaller arguments.

Consider the addition of ordinal numbers implemented in SML as follows:

```
datatype Ord = O'
             | S'  of Ord
             | Lim of Nat -> Ord

fun addord x O'      = x
  | addord x (S' y') = S' (addord x y')
  | addord x (Lim f) = Lim (fn z:Nat => addord x (f z))
```

`addord` is structurally recursive, since in each recursive call the second argument y is decreasing. In the first call `y' < S' y'` $=: y$ following axiom (1), and in the second call `f z` $\leq$ `f < Lim f` $=: y$ using both axioms.

We shall present syntactic conditions which are sufficient to ensure that a function is structurally recursive and which can be checked mechanically. We have implemented a termination checker for our language foetus that accepts structural recursive definitions. Recently this termination checker has been reimplemented by C. Coquand (1999) as part of Adga.

1.3 Lexicographic Termination Orderings

The foetus termination checker accepts also functions that are structural recursive w.r.t. a lexicographic extension of the structural ordering, e.g., the Ackermann function:

[1] E.g. this is implemented in Hugs, see (Jones & Reid, 1999), section 7.2.

```
fun ack 0      y      = S y
  | ack (S x') 0      = ack x' (S 0)
  | ack (S x') (S y') = ack x' (ack (S x') y')
```

Here we have to check three recursive calls: In `ack x' (S 0)` and `ack x'` ... the first argument is decreasing and in the third (nested) call `ack (S x') y'` the first argument stays the same and the second is decreased w.r.t. the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$.

1.4 Mutual Recursion and Call Graphs

To handle mutual recursion as well requires some additional considerations. We shall sketch our method by the means of the following example:

```
fun flat []          = []
  | flat (l::ls)     = aux l ls
and aux  []       ls = flat ls
  | aux  (x::xs) ls = x :: aux xs ls
```

The function `flat` takes a list of lists and returns a list making use of the auxiliary function `aux` which processes one list. We can extract three calls and the respective behavior of the arguments and organize them in the following *call graph* (solid arrows):

$$\begin{tikzcd} \mathrm{flat} \arrow[loop left, dotted, "(<)"] \arrow[rr, bend left, "\begin{pmatrix} < \\ < \end{pmatrix}"] & & \mathrm{aux} \arrow[ll, bend left, "\begin{pmatrix} ? & \leq \end{pmatrix}"] \arrow[loop right, "\begin{pmatrix} < & ? \\ ? & \leq \end{pmatrix}"] \end{tikzcd}$$

Each arrow is labeled with the *call matrix* of the respective call where the rows represent the input parameters of the calling function and the columns the arguments of the called function. A "$<$" in cell (i, j) of a call matrix expresses that the j argument of the call is strictly smaller than the ith parameter of the calling function, a "$\leq$" denotes weak decrease and a "?" stands for increase or absence of information. For example, in the call `flat`$\rightarrow$`aux` both arguments `l` and `ls` are structurally smaller than the input parameter `(l::ls)` of `flat`. By continued combination of adjacent calls we obtain the *completion* of that graph which contains the direct of indirect calls of a function to itself. In our case this yields for `flat` (dotted arrow):

$$\texttt{flat} \rightarrow \texttt{flat} : (<)$$

For a given function to be structural recursive, we now can apply the default demand: Each recursive call must decrease the argument structurally (which is given in the case of `flat`). We will treat call graphs formally in Sect. 3.

1.5 Semantic Analysis

The termination checker recognizes[2] structurally recursive functions. We want to know whether we can rely on the checker output, i.e., we want to know whether all structurally recursive functions terminate on all inputs. This is the case if the structural ordering is well-founded, which is a fundamental assumption in (Coquand, 1992). In Sect. 4 we give an interpretation of the types of our systems as monotone operators on value sets. We further introduce a structural ordering on all values in the interpretation of closed types and *prove* that this ordering is well-founded. Then we can show that all accepted terms normalize w.r.t. our operational semantics.

1.6 A Predicative Meta-Theory

In Sect. 4 we give a semantic interpretation of all types using an *impredicative* metatheory: We show that types which depend only strictly positively on type variables give rise to monotone operators on values. To define the interpretation of recursively defined types $\mu X.\sigma(X)$ we use the theorem by Knaster and Tarski (1955) that *every monotone operator on a complete lattice has a least fixpoint.* This construction can be extended to types which depend only positively on type variables (i.e., occuring in the left hand side of the scope of an even number of arrow types) because they give rise to monotone operators semantically (Abel, 1999).

To construct the least fixpoint of a monotone operator we use the completeness of the subset lattice, i.e., we construct the fixpoint as the intersection of all prefixpoints. This construction is impredicative—we define a new subset by quantifying over all subsets of an infinite set. In contrast, in a *predicative*[3] construction we only refer to concepts which have been defined previously. Thus we can construct fixpoints of strictly positive operators because they correspond to well-founded (but possibly infinitely branching) derivation trees. The induction principle can be justified because we refer to smaller (i.e., previously constructed trees) when constructing new trees. Examples of predicative theories are Martin-Löf's *Type Theory* (Martin-Löf, 1984; Nordström *et al.*, 1990) or Aczel's *Constructive Set Theory* (Aczel, 1997).

In Sect. 5 we show that it is possible to interpret our predicative type system in a predicative meta-theory. We use the concept of *set based operators* introduced by Peter Aczel (1997) in the context of intuitionistic set theory, actually here we need only the special case of deterministically set based operators. Intuitively, a set based operator can be understood as a monotone operator Φ which comes with a *urelement* relation $\mathcal{U}$. In the case of simple data structures like lists and trees over

[2] Note that the property *structurally recursive* is actually undecidable, hence we cannot hope for a complete decision procedure.

[3] Predicativity is not used in the sense of proof theory which calls theories which are stronger than Γ_0 impredicative. We consider all "bottom-up" theories as predicative, thus for instance also $ID^i_{<\omega}$.

a type τ this relation coincides with the "element"-relation. The urelement relation $\mathcal{U}$ enjoys the following properties:

1. If starting from a value set V we construct a value set $\Phi(V)$ with an element $w \in \Phi(V)$, then all urelements $v\ \mathcal{U}\ w$ of w are elements of the base set V.
2. An element $w \in \Phi(\text{True})$ (where True is the universal value set) can be reconstructed from its urelements, i.e., $w \in \Phi(\{v : v\ \mathcal{U}\ w\})$.

We show that every strictly positive type can be interpreted by a set based operator and that fixpoints of set based operators can be constructed by strictly positive inductive definitions on the meta level. This idea has already been used in a predicative strong normalisation proof (Abel & Altenkirch, 2000).

1.7 Notational conventions

We are using vectors to simplify our notation. If we have a family of expressions $E_1, E_2, \ldots, E_n$ we write $\vec{E}$ for the whole sequence. We denote the length n of the sequence by $|\vec{E}|$. Given a fixed E' we write $\vec{E}E'$ for E_1E', E_2E', …, E_nE' and $E'\vec{E}$ for $E'E_1$, $E'E_2$, … $E'E_n$. Furthermore we use the notations

X, Y, Z	for	type variables
ρ, σ, τ	for	types
x, y, z, l	for	term variables
r, s, t, a	for	terms
u, v, w	for	values
f, g	for	function identifiers or values
c	for	closures
e	for	environments

We use set notation to define predicates, i.e., we write $x \in P$ for $P(x)$ and we define new predicates by the notation for set comprehension. However, sets are not first order citizens in our meta theory, i.e., we do not quantify over sets and we do not use power sets. We write relations in infix notation, i.e., we write $x\ R\ y$ for $(x, y) \in R$. We write projections as partial applications, i.e., $R(y) = \{x : x\ R\ y\}$.

2 The foetus Language

We introduce a term language based on the simply typed lambda calculus enriched with finite sum and products and inductive types, in the style of a functional programming language. Thus we allow the definition of recursive terms. We define normal forms explicitely as *values* and introduce environments avoiding to have to define substitution. The meaning of terms is defined by a big step operational semantics.

2.1 Types

The foetus type system is constructed from the base type variables $X, Y, Z, \ldots$ and the type constructors Σ (finite sum), Π (finite product), $\to$ (function space) and μ

(inductive type). We restrict ourselves to strictly positive recursive types for two reasons: Firstly, it is not clear how to formalize structural recursion for non-strictly positive types. And secondly, it is not clear whether non-strictly positive types can be understood predicatively. Within our system the set of natural numbers may be expressed as $\mathbf{N} \equiv \mu X.1 + X$ and the set of lists over natural numbers as $\mathbf{L}^{\mathbf{N}} \equiv \mu X.1 + \mathbf{N} \times X$.

Type Variables. Assume a countably infinite set of type variables TyVar whose elements we denote by X, Y, Z, ... A type variable X appears *strictly positive* within a type τ iff it appears never on the left side of some $\rightarrow$. We enforce that all variables in a type satisfy this condition by restricting the domains of function types to closed types (see rule (Arr) below).

Definition 2.1 (*Types*)
We inductively define the family of types $\mathrm{Ty}(\vec{X})$ indexed over a finite list of distinct type variables $\vec{X} \in \mathrm{TyVar}$ appearing only strictly positive as follows:

$$(\mathrm{Var})\ \frac{X_1, \ldots, X_n \in \mathrm{TyVar} \quad 1 \leq i \leq n}{X_i \in \mathrm{Ty}(\vec{X})}$$

$$(\mathrm{Sum})\ \frac{\sigma_1, \ldots, \sigma_n \in \mathrm{Ty}(\vec{X})}{\Sigma\vec{\sigma} \in \mathrm{Ty}(\vec{X})} \qquad (\mathrm{Prod})\ \frac{\sigma_1, \ldots, \sigma_n \in \mathrm{Ty}(\vec{X})}{\Pi\vec{\sigma} \in \mathrm{Ty}(\vec{X})}$$

$$(\mathrm{Arr})\ \frac{\sigma \in \mathrm{Ty}() \quad \tau \in \mathrm{Ty}(\vec{X})}{\sigma \rightarrow \tau \in \mathrm{Ty}(\vec{X})} \qquad (\mathrm{Mu})\ \frac{\sigma \in \mathrm{Ty}(\vec{X}, X)}{\mu X.\sigma \in \mathrm{Ty}(\vec{X})}$$

We can restrict the free type variables to strictly positive ones because unlike Girard's System F (Girard, 1972), we have no polymorphic types and therefore need type variables only to construct inductive 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 Ty() by Ty. For binary sums $\Sigma(\sigma, \tau)$ we write $\sigma + \tau$, for ternary $\rho + \sigma + \tau$, etc. (same for products). The empty sum $\Sigma()$ is denoted by 0 and the empty product $\Pi()$ by 1.

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

Our style of variable introduction and binding (see rules (Var) and (Mu) below) is very close to an implementation of variables by deBruijn-indices (see (de Bruijn, 1972)), but we have kept variable names for better readability.

Substitution. Provided a type $\sigma(\vec{X})$ with a list of free variables $\vec{X}$ and an equally long list of types $\vec{\rho} \in \mathrm{Ty}(\vec{Y})$ we define the capture avoiding substitution $\sigma[\vec{X} := \vec{\rho}] \in$

$\mathrm{Ty}(\vec{Y})$ in the usual way. 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 types defined above. 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 using contexts to simplify the definition of closures afterwards.

Term Variables and Contexts. Assume a countably infinite set of term variables TmVar using g, x, y, z to denote elements of this set. Given 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 assumption that the variable is of type σ.

Definition 2.2 (*Terms*)
We define the set of (well-typed) terms $\mathrm{Tm}^\sigma[\Gamma]$ of a closed type σ in context Γ inductively as follows:

$$(\mathrm{var})\ \frac{\Gamma, x^\sigma, \Gamma' \in \mathrm{Cxt}}{x \in \mathrm{Tm}^\sigma[\Gamma, x^\sigma, \Gamma']}$$

$$(\mathrm{in})\ \frac{t \in \mathrm{Tm}^{\sigma_j}[\Gamma]}{\mathrm{in}_j^{\vec{\sigma}}(t) \in \mathrm{Tm}^{\Sigma\vec{\sigma}}[\Gamma]}$$

$$(\mathrm{case})\ \frac{t \in \mathrm{Tm}^{\Sigma\vec{\sigma}}[\Gamma] \qquad t_i \in \mathrm{Tm}^\rho[\Gamma, x_i^{\sigma_i}] \text{ for } 1 \leq i \leq n}{\mathrm{case}(t, x_1^{\sigma_1}.t_1, \ldots, x_n^{\sigma_n}.t_n) \in \mathrm{Tm}^\rho[\Gamma]}$$

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

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

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

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

There are two main kinds of term forming rules: Rules for introducing types (the constructors (in), (tup), (lam) and (fold)) and rules for eliminating types (the destructors (case), (pi), (app) and (unfold)). The remaining rules (var) and (rec) are structural.

Renaming Convention for Terms. In these rules case binds the variables x_i in the terms t_i, λ binds x in t and rec binds g in t. As for 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 $\mathrm{Tm}^\sigma[]$. We omit type annotation on in_j, fold, unfold and abstractions where ever we do not introduce ambiguity.

Currying. For our termination analysis it is convenient to view a function with several arguments in its uncurried form as a function with one argument of product type. To support the formulation of functions of arity ≥ 2, we introduce the construct

$$\frac{t \in \mathrm{Tm}^\tau[\Gamma, x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}]}{\lambda(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}).\, t \in \mathrm{Tm}^{\Pi\vec{\sigma}\to\tau}[\Gamma]}$$

From the semantic point of view it is just syntactic sugar for $\lambda y^{\Pi\vec{\sigma}}.\, t'$ where t' is the result of replacing x_i with $\pi_i(y)$ for $1 \leq i \leq n$. For instance,

$$\lambda(x^\sigma, y^\tau).\,(y, x) \qquad \text{is interpreted as} \qquad \lambda p^{\sigma\times\tau}.\,(\pi_2(p), \pi_1(p)).$$

2.2.1 Example: The `flat` *Function*

To clarify the different constructs of our language we give an encoding of the list flattening function in foetus. This function takes a list of lists of natural numbers and transforms it into a list of natural numbers. All required datatypes can be defined in foetus using fixpoint types:

$$\begin{aligned}
\mathbf{Nat} &\equiv \mu X.\, 1 + X \\
\mathbf{ListN} &\equiv \mu X.\, 1 + \mathbf{Nat} \times X \\
\mathbf{ListL} &\equiv \mu X.\, 1 + \mathbf{ListN} \times X
\end{aligned}$$

Identifiers in **bold** font denote defined types or terms which are not part of the language but mere abbreviations. For convenience, we define abbreviations for the two constructors of lists:

$$\begin{aligned}
\mathbf{nil} &\equiv \mathrm{fold}(\mathrm{in}_1()) \\
\mathbf{cons}(-) &\equiv \mathrm{fold}(\mathrm{in}_2(-))
\end{aligned}$$

For a natural number $x \in \mathrm{Tm}^{\mathbf{Nat}}$ and a list $y \in \mathrm{Tm}^{\mathbf{ListN}}$ we can construct the extension $\mathbf{cons}(x, y)$ of list y by element x as follows:

$$\cfrac{\cfrac{\cfrac{x \in \mathrm{Tm}^{\mathbf{Nat}} \qquad y \in \mathrm{Tm}^{\mathbf{ListN}}}{(x,y) \in \mathrm{Tm}^{\mathbf{Nat}\times\mathbf{ListN}}}\,\mathrm{tup}}{\mathrm{in}_2(x,y) \in \mathrm{Tm}^{1+\mathbf{Nat}\times\mathbf{ListN}}}\,\mathrm{in}}{\mathrm{fold}^{X.\,1+\mathbf{Nat}\times X}(\mathrm{in}_2(x,y)) \in \mathrm{Tm}^{\mu X.\,1+\mathbf{Nat}\times X}}\,\mathrm{fold}$$

Using the defined abbreviations the final line states $\mathbf{cons}(x, y) \in \mathrm{Tm}^{\mathbf{ListN}}$. We recall the definition of the flattening function:

```
fun flat []          = []
  | flat (l::ls)     = aux l ls
and aux   []      ls = flat ls
  | aux   (x::xs) ls = x :: aux xs ls
```

The main function `flat` is recursive, taking a list of lists and returning a list of natural numbers. We will denote it with the identifier f of type $\mathbf{ListL} \to \mathbf{ListN}$. The pattern matching over its one argument, which we will refer to as ll, translates into a case construct:

$$\begin{aligned}\mathbf{flat} \equiv\ & \mathrm{rec}\, f^{\mathbf{ListL}\to\mathbf{ListN}}.\, \lambda \mathit{ll}^{\mathbf{ListL}}.\\ & \mathrm{case}(\mathrm{unfold}(\mathit{ll}),\\ & \qquad \_^{1}.\,\mathbf{nil},\\ & \qquad p^{\mathbf{ListN}\times\mathbf{ListL}}.\,\mathbf{aux}\,(\pi_1(p), \pi_2(p)))\end{aligned}$$

Since ll is of fixpoint type we first have to unfold it to obtain something of sum type before we can distinct cases. The first alternative is that $\mathrm{unfold}(\mathit{ll}) = \mathrm{in}_1(\_^1)$, which means that it is an injection of something of unit type 1. Since we are not interested in the specific inhabitant of the unit type involved here, we use the anonymous variable "_". In this case ll encodes the empty list, and we return the empty list **nil**. In the second case $\mathrm{unfold}(\mathit{ll}) = \mathrm{in}_2(p^{\mathbf{ListN}\times\mathbf{ListL}})$ we have to deal with the injection of a pair p consisting of a list of natural numbers and a list of lists. This case is handled by the auxiliary function, denoted by the placeholder **aux**.

The *simultaneous* recursion of SML can be translated into a *nested* or *interleaving* recursion in `foetus`. We simply define `aux` via the recursion operator rec and substitute it for every occurrence of the placeholder **aux** in **flat**. In this way we realize calls from `flat` to `aux`. All calls from `aux` to `flat` can simply refer to the identifier f, since **aux** is within the scope of f. Omitting some type annotations the full program reads as follows:

$$\begin{aligned}\mathbf{flat} \equiv\ & \mathrm{rec}\, f.\, \lambda \mathit{ll}.\\ & \mathrm{case}(\mathrm{unfold}(\mathit{ll}),\\ & \qquad \_.\,\mathbf{nil},\\ & \qquad p.\,(\mathrm{rec}\, g.\, \lambda(l^{\mathbf{ListN}}, \mathit{ls}^{\mathbf{ListL}}).\\ & \qquad\quad \mathrm{case}(\mathrm{unfold}(l),\\ & \qquad\qquad \_.\, f\, \mathit{ls},\end{aligned}$$

$$q^{\mathbf{Nat}\times\mathbf{ListN}}.\,\mathbf{cons}(\pi_1(q),\ g(\pi_2(q), ls))))$$
$$(\pi_1(p), \pi_2(p)))$$

For the definition of **aux**—which we assigned function identifier g—we used the syntactic sugar. To obtain the unsugared version we replace

$$\begin{array}{ccc} (l^{\mathbf{ListN}}, ls^{\mathbf{ListL}}) & \text{with} & y^{\mathbf{ListN}\times\mathbf{ListL}}, \\ l & \text{with} & \pi_1(y) \quad \text{and} \\ ls & \text{with} & \pi_2(y). \end{array}$$

Readers familiar with SML will know that `fun` is not a core construct of the programming language and can be replaced by the more primitive `val rec` (Harper, 2000). Replacing simultaneous recursion with interleaving recursion, we obtain a version of the original program which is very similar to its foetus encoding:

```
val rec flat = fn ll =>
    let val rec aux = fn (l, ls) =>
        case l of
            [] => flat ls
          | (x::xs) => x :: aux (xs, ls)
    in
        case ll of
            [] => []
          | (l::ls) => aux (l, ls)
    end
```

Note that `rec` has to be used in conjunction with a `let` in SML, whereas it can appear in any position in foetus programs.

2.3 Values and Closures

We only assign a meaning to closed terms $t \in \mathrm{Tm}^\sigma$: they evaluate to *(syntactic) values* of type σ. (We are going to define *semantic* values as well, see Sect. 4.) During the process of evaluation defined by our operational semantics we will have to handle open terms, together with environments which assign values to the free variables. This entails that we do not have to define substitution on terms. A Term and its corresponding environment form a *closure* which can be seen as completion of an open term.

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).

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

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

$$\text{(vin)}\ \frac{v \in \mathrm{Val}^{\sigma_j}}{\mathrm{in}_j^{\vec\sigma}(v) \in \mathrm{Val}^{\Sigma\vec\sigma}} \qquad \text{(vtup)}\ \frac{v_i \in \mathrm{Val}^{\sigma_i} \text{ for } 1 \le i \le n}{(v_1, \ldots, v_n) \in \mathrm{Val}^{\Pi\vec\sigma}}$$

$$\text{(vfold)}\ \frac{v \in \mathrm{Val}^{\sigma(\mu X.\sigma)}}{\mathrm{fold}^{X.\sigma}(v) \in \mathrm{Val}^{\mu X.\sigma}}$$

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

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 value) are convenient to define the operational semantics without casting 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 that differ only in variable names as equal.

2.4 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.

Definition 2.3 (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.

$$\text{(opvar)}\ \frac{}{\langle x; e, x = v, e'\rangle \Downarrow v}$$

$$\text{(opin)}\quad \frac{\langle t; e\rangle \Downarrow v}{\langle \mathrm{in}_j(t); e\rangle \Downarrow \mathrm{in}_j(v)}$$

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

$$\text{(optup)}\quad \frac{\langle t_i; e\rangle \Downarrow v_i \text{ for } 1 \leq i \leq n}{\langle (\vec{t}); e\rangle \Downarrow (\vec{v})} \qquad \text{(oppi)}\quad \frac{\langle t; e\rangle \Downarrow (\vec{v})}{\langle \pi_j(t); e\rangle \Downarrow v_j}$$

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

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

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

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

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

Proposition 2.4
$\Downarrow$ is deterministic, i.e., $c \Downarrow v$ and $c \Downarrow v'$ implies $v = v'$.

Proof
For all closures except $\langle \mathrm{case}(t, \vec{x.t}); \ldots\rangle$ there is only one computation rule. But also for closures with case analysis there will be only one computation tree, since $\langle t; \ldots\rangle$ may only evaluate to $\mathrm{in}_j(w)$ for a fixed j thus only one instance of (opcase) is applicable. □

3 The foetus Termination Checker

The syntactic check, whether a recursive term is structurally recursive, consists of three phases:

1. Function call extraction.
   The recursive term is analysed and all recursive calls are gathered. During the analysis dependencies of new variables are gained, i.e., information whether a new variable is structurally smaller than an already known one.
2. Call graph generation and completion.
   All calls which have been gathered are organized in a call graph which is completed afterwards.

3. Search for a termination ordering.
   From the completed graph all recursive calls of the function in consideration are extracted. Then it is searched for a lexicographic ordering on the arguments that ensures termination.

The termination checker makes use of two major data structures, *dependencies* and *calls*, and a minor data structure, the *function stack*.

Dependencies Δ: A set of dependencies Δ is a collection of straitened variables $x \leq s$. We use these dependencies to derive that arguments decrease structurally in a recursive call. They are generated whenever we learn information about terms when stepping over a case construct. The semantics of the term

$$\mathrm{case}(s, x_1.t_1, \ldots, x_n.t_n)$$

yields the constraints $\mathrm{in}_j(x_j) = s$ for $1 \leq j \leq n$. Thus within the scope of x_j, i.e, within t_j, we may safely assume that $x_j \leq s$.

Calls $\mathcal{C}$: A call $g(t_1, \ldots, t_n)$ of function g with arguments $\vec{t}$ within a function f with formal parameters $(x_1, \ldots, x_m)$ is denoted by

$$f \to g : (a_{ij}) \quad \text{where } a_{ij} \in \{<, \leq, ?\} \text{ for } 1 \leq i \leq n, 1 \leq j \leq m$$

We will refer to (a_{ij}) as *call matrix*, since its elements indicate how the arguments $\vec{t}$ behave in a function call: $a_{ij} = <$ if t_i can be shown to always be strictly structurally smaller than x_j, $a_{ij} = \leq$ if t_i is ensure to be less or equal than x_j, and $a_{ij} = ?$ if neither of the previous alternatives holds. We denote sets of calls by $\mathcal{C}$.

Function stack Φ: As we traverse a term we build up a function stack Φ which contains the name of functions together with their formal arguments. When we analyse

$$\mathrm{rec}\, f.\, \lambda(x_1, \ldots, x_n).\, s$$

we will push $f(x_1, \ldots, x_n)$ onto the stack. The topmost element denotes the function whose body we are currently inspecting.

The procedure which analyzes a term, finds all function calls and computes the call matrices will be presented in form of two judgments:

$\Delta, \Phi \vdash t \sqsupset \mathcal{C}$	function call extraction
$\Delta \vdash t \; R \; x \; (R \in \{<, \leq, ?\})$	structural ordering on terms

In the remainder of this section we will define these two judgments (Sect. 3.1 and 3.2), state some properties about call matrices (Sect. 3.3), give an algorithm that finds indirect calls via graph completion (Sect. 3.4) and show how to find a lexicographic ordering that ensures termination (Sect. 3.5).

3.1 Function Call Extraction

Given a foetus program t the termination checker first extracts the set of function calls $\mathcal{C}$ within t. The extraction process can be formalized as follows:

Definition 3.1 (*Function Call Extraction*)
Let Δ be a set of dependencies, Φ a function stack, t a term and $\mathcal{C}$ a set of calls. Then $\mathcal{C}$ is extracted from t iff

$$\Delta; \Phi \vdash t \sqsupset \mathcal{C}$$

Read "under the dependencies Δ and within the functions Φ the term t contains the calls $\mathcal{C}$". The judgment is established by the following rules:

Recursive functions and calls:

$$(\sqsupset\text{rec})\ \frac{\Delta; \Phi, g(x_1, \dots, x_n) \vdash s \sqsupset \mathcal{C}}{\Delta; \Phi \vdash \mathrm{rec}\, g.\, \lambda(x_1, \dots x_n).\, s \sqsupset \mathcal{C}}$$

$$(\sqsupset\text{appvar})\ \frac{\Delta; \Phi, f(\vec{x}) \vdash (\vec{t}) \sqsupset \mathcal{C} \qquad \Delta \vdash t_i\ a_{ij}\ x_j \quad \text{for all } 1 \leq i \leq n, 1 \leq j \leq m}{\Delta; \Phi, f(x_1, \dots, x_m) \vdash g\,(t_1, \dots, t_n)\ \sqsupset\ \mathcal{C}, f \to g : (a_{ij})}$$

$$(\sqsupset\text{apprec})\ \frac{\Delta; \Phi \vdash (\vec{t}) \sqsupset \mathcal{C} \qquad \Delta; \Phi, g(\vec{y}) \vdash s \sqsupset \mathcal{C}' \qquad \Delta \vdash t_i\ a_{ij}\ x_j \quad \text{for all } i, j}{\Delta; \Phi \equiv (\Phi', f(\vec{x})) \vdash (\mathrm{rec}\, g.\, \lambda(\vec{y}).\, s)\ (\vec{t})\ \sqsupset\ \mathcal{C}, \mathcal{C}', f \to g : (a_{ij})}$$

Dependencies:

$$(\sqsupset\text{case})\ \frac{\Delta; \Phi \vdash s \sqsupset \mathcal{C} \qquad \Delta, x_i \leq s; \Phi \vdash t_i \sqsupset \mathcal{C}_i \quad \text{for all } i}{\Delta; \Phi \vdash \mathrm{case}(s, \vec{x.t}) \sqsupset \mathcal{C}, \vec{\mathcal{C}}}$$

Congruences:

$$(\sqsupset\text{var})\ \frac{}{\Delta; \Phi \vdash x \sqsupset \cdot} \qquad (\sqsupset\text{in})\ \frac{\Delta; \Phi \vdash t \sqsupset \mathcal{C}}{\Delta; \Phi \vdash \mathrm{in}_j(t) \sqsupset \mathcal{C}}$$

$$(\sqsupset\text{tup})\ \frac{\Delta; \Phi \vdash t_i \sqsupset \mathcal{C}_i \quad \text{for all } i}{\Delta; \Phi \vdash (\vec{t}) \sqsupset \vec{\mathcal{C}}} \qquad (\sqsupset\text{pi})\ \frac{\Delta; \Phi \vdash t \sqsupset \mathcal{C}}{\Delta; \Phi \vdash \pi_j(t) \sqsupset \mathcal{C}}$$

$$(\sqsupset\text{lam})\ \frac{\Delta; \Phi \vdash t \sqsupset \mathcal{C}}{\Delta; \Phi \vdash \lambda x.\, t \sqsupset \mathcal{C}} \qquad (\sqsupset\text{app})\ \frac{\Delta; \Phi \vdash t \sqsupset \mathcal{C} \qquad \Delta; \Phi \vdash s \sqsupset \mathcal{C}'}{\Delta; \Phi \vdash t\, s \sqsupset \mathcal{C}, \mathcal{C}'}$$

$$(\sqsupset\text{fold})\ \frac{\Delta; \Phi \vdash t \sqsupset \mathcal{C}}{\Delta; \Phi \vdash \mathrm{fold}(t) \sqsupset \mathcal{C}} \qquad (\sqsupset\text{unfold})\ \frac{\Delta; \Phi \vdash t \sqsupset \mathcal{C}}{\Delta; \Phi \vdash \mathrm{unfold}(t) \sqsupset \mathcal{C}}$$

Note that only the rules ($\sqsupset$appvar) and ($\sqsupset$apprec) add new calls to the set $\mathcal{C}$. To calculate the call matrix (a_{ij}) they rely on the judgment infering structural ordering on terms, which we will describe in the next section.

Read upwards the given rules provide an algorithm for extracting function calls out of a well-typed term t. As stated, it is non-deterministic since several rules for application exist. In the following we stipulate that ($\sqsupset$appvar) and ($\sqsupset$apprec) always override ($\sqsupset$app). This makes the algorithm deterministic and complete, i.e., it will return all function calls to be found in t.

Let us clarify the algorithm by an example. Recall the flattening function and its implementation in foetus (now stripped of all type annotations):

$$\begin{aligned}
\mathbf{flat} \equiv \operatorname{rec} f.\, \lambda ll.\, & \operatorname{case}(\operatorname{unfold}(ll), \\
& _.\, \mathbf{nil}, \\
& p.\, (\operatorname{rec} g.\, \lambda(l, ls).\, \operatorname{case}(\operatorname{unfold}(l), \\
& \qquad _.\, f\, ls, \\
& \qquad q.\, \mathbf{cons}(\pi_1(q),\ g\,(\pi_2(q), ls)))) \\
& \quad (\pi_1(p), \pi_2(p)))
\end{aligned}$$

The following table gives snapshots of the extraction algorithm in action. We display the dependencies Δ and the function stack Φ at the points where calls are detected.

Δ	Φ	t	call
$\Delta_0 \equiv p \leq \operatorname{unfold}(ll)$	$f(ll)$	$(\operatorname{rec} g.\, \lambda(l, ls) \ldots)\,(\pi_1(p), (\pi_2(p))$	$f \to g$
$\Delta_0, q \leq \operatorname{unfold}(l)$	$f(ll), g(l, ls)$	$f\, ls$	$g \to f$
$\Delta_0, q \leq \operatorname{unfold}(l)$	$f(ll), g(l, ls)$	$g\,(\pi_2(q), ls)$	$g \to g$

In all three cases the call matrizes can be computed from Δ, Φ and the arguments of the call. How, we will present in the next section.

3.2 Structural Ordering on Terms

The second integral part of the termination checker is a calculus that allows us to determine whether from a set of dependencies Δ we can derive a relation p of the form $t < x$ or $t \leq x$. Since its use is restricted to calculate whether a function argument t is structurally smaller than a function parameter x, the right hand side will always be a variable.

Definition 3.2 (*Structural Ordering on Terms*)
Let Δ be a set of dependencies, t a term and x a variable. The judgment

$$\Delta \vdash t\ R\ x \qquad R \in \{<, \leq\}$$

states that "under the dependecies Δ the term s is (strictly) structurally smaller than the variable x". It is defined by the rules given after the following motivation.

As stated in the introduction, we motivate structural ordering mainly by the fact that a constructor of an inductive datatype increases the order. For example,

$$l < \mathbf{cons}(x, l) \equiv \operatorname{fold}(\operatorname{in}_2(x, l)) \tag{3}$$

Since every constructor of an inductive datatype involves a fold in the outermost position, it is sufficient for the other term constructors like pairing and injection to increase the order weakly. Thus we can form the chain

$$l \leq (x, l) \leq \operatorname{in}_2(x, l) < \operatorname{fold}(\operatorname{in}_2(x, l))$$

This term *constructor* oriented definition is motivated by the observable *values* of the programming language. We will come back to it in Sect. 4.2. However, for the program analysis we perform, we have to consider the term *destructors*. Consider the recursive function

$$\mathrm{rec}\, f.\, \lambda x \ldots f\, t \ldots$$

The argument t can be shown to be structurally smaller than the parameter x if it is derived from x involving only destructors.[4] From the perspective of destructors the inequality (3) looks as follows ($l \neq \mathbf{nil}$):

$$\mathbf{tail}(l) \equiv \mathrm{case}(l,\ \_.\mathbf{nil},\ p.\pi_2(p)) < l \tag{4}$$

This motivates the following rules ($R \in \{<, \leq\}$):

$$(\leq\mathrm{refl})\ \frac{}{\Delta \vdash x \leq x}$$

$$(R\mathrm{case})\ \frac{\Delta, y_i \leq s \vdash t_i\ R\ x \text{ for } i=1,\ldots,n}{\Delta \vdash \mathrm{case}(s, \vec{y.t})\ R\ x} \qquad (R\mathrm{pi})\ \frac{\Delta \vdash t\ R\ x}{\Delta \vdash \pi_j(t)\ R\ x}$$

$$(R\mathrm{app})\ \frac{\Delta \vdash f\ R\ x}{\Delta \vdash f\, a\ R\ x} \qquad (R\mathrm{unf})\ \frac{\Delta \vdash t \leq x}{\Delta \vdash \mathrm{unfold}(t)\ R\ x}$$

During the process of function call extraction, whenever we step into a branch t of a case construct $\mathrm{case}(s, \ldots, y.t, \ldots)$ we introduce a the new variable y which is constrained by the dependency $y \leq s$. In checking $\Delta \vdash t\ R\ x$, whenever we have reduced t to a variable y by the destructor rules above, we can use the dependency of y by transitivity ($R \in \{<, \leq\}$).

$$(R\mathrm{trans})\ \frac{\Delta, y \leq t, \Delta' \vdash t\ R\ x}{\Delta, y \leq t, \Delta' \vdash y\ R\ x}$$

The arising calculus is deterministic. The number of rule applications necessary to decide $\Delta \vdash t\ R\ x$ is limited by the size of Δ, i.e., the sum of the sizes of all terms in Δ.

Continuing our example from the last section we show that in the call $f \to g$ the second argument to g is smaller than the parameter of f, i.e., $p \leq \mathrm{unfold}(ll) \vdash \pi_2(p) < ll$). Furthermore we show for the call $g \to g$ that $q \leq \mathrm{unfold}(l) \vdash \pi_2(q) < l$ which means that the first argument is decreased.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash ll \leq ll}\leq\mathrm{refl}}{\ldots \vdash \mathrm{unfold}(ll) < ll}<\mathrm{unf}}{p \leq \mathrm{unfold}(ll) \vdash p < ll}<\mathrm{trans}}{p \leq \mathrm{unfold}(ll) \vdash \pi_2(p) < ll}<\mathrm{pi} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash l \leq l}\leq\mathrm{refl}}{\ldots \vdash \mathrm{unfold}(l) < l}<\mathrm{unf}}{q \leq \mathrm{unfold}(l) \vdash q < l}<\mathrm{trans}}{q \leq \mathrm{unfold}(l) \vdash \pi_2(q) < l}<\mathrm{pi}$$

[4] For the sake of simplicity we ignore the fact that constructors might be contained in t if they are eliminated by a larger number of destructors. See Remark 3.3.

Remark 3.3 (*Limitations*)
Since we limit the rules to destructors, the termination checker does not accept recursive calls with constructors in the arguments. A consequence of that is that we do not recognize termination of functions like the following, which computes the sum of a list:

```
fun sum []           = 0
  | sum (S n :: l) = S (sum (n :: l))
  | sum (0 :: l)   = sum l
```

However, this program can be mechanically transformed into one that terminates by a lexicographic ordering.

Remark 3.4 (*Deciding R*)
Given Δ, t and x the calculus can also decide whether $\Delta \vdash t < x$ or only $\Delta \vdash t \leq x$. For that purpose we construct a derivation for $\Delta \vdash t\ R\ x$ where R is undetermined. The first time we invoke the rule for unfold we fix $R = <$.

Definition 3.5
We write $\Delta \vdash t\ ?\ x$ iff *not* $\Delta \vdash t\ R\ x$ for $R \in \{<, \leq\}$.

3.3 Call Matrices

The behavior of the arguments in a call $f(x_1, \ldots, x_m) \to g(t_1, \ldots, t_n)$ with dependencies Δ can be expressed by a matrix $A = (a_{ij}) \in \mathcal{R}^{n \times m}$, where $\mathcal{R} = \{<, \leq, ?\}$. We define

$$a_{ij} = \begin{cases} < & \text{if } \Delta \vdash t_i < x_j \\ \leq & \text{if } \Delta \vdash t_i \leq x_j, \text{ but not } \Delta \vdash t_i < x_j \\ ? & \text{if not } \Delta \vdash t_i \leq x_j \end{cases}$$

The columns of that matrix stand for the input variables of the calling function and the rows for the arguments of the call. Together with the two operations $+$ and $\cdot$ defined as in the table below the set $\mathcal{R}$ forms a commutative *semi-ring* with zero $?$ and unit $\leq$.

$+$	$<$	$\leq$	$?$
$<$	$<$	$<$	$<$
$\leq$	$<$	$\leq$	$\leq$
$?$	$<$	$\leq$	$?$

$\cdot$	$<$	$\leq$	$?$
$<$	$<$	$<$	$?$
$\leq$	$<$	$\leq$	$?$
$?$	$?$	$?$	$?$

The operation $+$ can be understood as "combining *parallel* information about a relation". For instance, if we have $a\ ?\ y$ and $a < y$ we have $a\,(? + <)\,y$ and that simplifies to $a < y$. The operation $\cdot$ however is "*serial* combination", for example, $a < y$ and $y \leq z$ can be combined into $a\,(< \cdot \leq)\,z$, simplified: $a < z$. The element $?$ is neutral regarding $+$ because it provides no new information, whereas $<$ is dominant because it is the strongest information. Regarding $\cdot$ the relation $\leq$ is neutral and $?$ is dominant because it "destroys" all information.

Since $\mathcal{R}$ is a semi-ring, we can define multiplication on matrices over $\mathcal{R}$ which allows us to compute the size change information of a *combined call* in the following sense: Given two calls

$$\begin{array}{rcl} f(x_1,\dots,x_l) & \to & g(y_1,\dots,y_m) \quad : \quad (a_{ij}) \in \mathcal{R}^{m\times l}, \text{ and} \\ g(y_1,\dots,y_m) & \to & h(z_1,\dots,z_n) \quad : \quad (b_{ij}) \in \mathcal{R}^{n\times m}, \end{array}$$

the argument behavior of the indirect call $f \to g \to h$ is captured by the matrix product

$$(c_{ij}) = (b_{ij})(a_{ij}) = \left(\sum_{k=1}^{m} b_{ik}a_{kj}\right) \in \mathcal{R}^{n\times l}.$$

Definition 3.6 (*Call Matrix*)
A *call matrix* is a matrix over $\mathcal{R}$ with no more than one element different from "?" per row.

$$\mathrm{CM}(n,m) := \{(a_{ij}) \in \mathcal{R}^{n\times m} : \forall i \forall j \forall k \neq j (a_{ij} = ? \text{ or } a_{ik} = ?)\}$$

Remark 3.7
The reason we define call matrices this way is these are the only ones foetus produces by function call extraction. By definiton of the structural ordering on terms a call argument can only depend on *one* function parameter.

For the three calls in our example the call matrices are

$$f \to g : \begin{pmatrix} < \\ < \end{pmatrix} \qquad g \to g : \begin{pmatrix} < & ? \\ ? & \leq \end{pmatrix} \qquad g \to f : \begin{pmatrix} ? & \leq \end{pmatrix}$$

The next proposition ensures that all matrices foetus will have to deal with are *call* matrices.

Proposition 3.8 (*Call Matrix Multiplication*)
Multiplication on matrices induces a multiplication on call matrices

$$\cdot : \mathrm{CM}(n,m) \times \mathrm{CM}(m,l) \to \mathrm{CM}(n,l)$$

This operation is well defined.

Proof
Let $A = (a_{ij}) \in \mathrm{CM}(n,m)$, $B = (b_{ij}) \in \mathrm{CM}(m,l)$, $AB = C = (c_{ij}) \in \mathcal{R}^{n\times l}$ and $k(i)$ the index of the element of the ith row of A that is different from "?" (or 1, if no such element exists). Then we have with the rules in semi-ring $\mathcal{R}$

$$c_{ij} = \sum_{k=1}^{m} a_{ik}b_{kj} = a_{i,k(i)}b_{k(i),j}$$

Now consider the ith row of C:

$$c_i = (c_{ij})_{1\leq j\leq l} = (a_{i,k(i)}b_{k(i),j})_{1\leq j\leq l}$$

Because at most one $b_{k(i),j}$ is different from "?", at most one element of c_i is different from "?". Therefore $C \in \mathrm{CM}(n,l)$. $\square$

3.4 Call Graphs

In the following we give a formal treatment of call graphs together with a graph completion algorithm which is a central part of the termination checker.

The nodes of call graphs are function identifiers with a given arity (the names of the function parameters are no longer of relevance). For each $i \in \mathbb{N}$ we assume an infinite supply $F^{(i)} = \{f^{(i)}, g^{(i)}, h^{(i)}, ...\}$ of identifiers for functions of arity i. The edges of a call graph are taken from the set of *calls*:

$$\text{Calls} = \{(f^{(n)}, g^{(m)}, A) : f^{(n)} \in F^{(n)}, g^{(m)} \in F^{(m)}, A \in \text{CM}(m, n)\}.$$

Since calls contain (besides the call matrix A) the source $(f^{(n)})$ and the target $(g^{(m)})$ function identifier with arities, we can simply identify a call graph with its edges:

Definition 3.9 (*Call Graph*)
A *call graph* is a finite set of calls $\mathcal{C} \subseteq \text{Calls}$.

By the following operation, two consecutive calls can be put together.

Definition 3.10 (*Call Combination*)
The partial operation *combination of calls* is given by

$$\circ : \text{Calls} \times \text{Calls} \to \text{Calls}$$
$$\Big((g^{(m)}, h^{(l)}, B), (f^{(n)}, g^{(m)}, A)\Big) \mapsto (f^{(n)}, h^{(l)}, BA).$$

That is, if g calls h with call matrix B and f calls g with call matrix A, then f indirectly calls h with call matrix BA. The operation $\circ$ cannot be applied to calls that have no "common function" like g, therefore it is partial. Call combination can be lifted to sets of calls

$$\circ : \mathcal{P}(\text{Calls}) \times \mathcal{P}(\text{Calls}) \to \mathcal{P}(\text{Calls})$$
$$(\mathcal{C}, \mathcal{C}') \mapsto \{c \circ_{\text{Calls}} c' : c \in \mathcal{C}, c' \in \mathcal{C}', (c, c') \in \text{Dom}(\circ_{\text{Calls}})\}$$

Here we combine each call in $\mathcal{C}$ with each call in $\mathcal{C}'$ to which $\circ_{\text{Calls}}$ is applicable and form a set of the combined calls. Of course, $\circ_{\mathcal{P}(\text{Calls})}$ is a total and monotonic function. For $n \geq 1$,

$$\mathcal{C}^n := \underbrace{\mathcal{C} \circ \ldots \circ \mathcal{C}}_{n}$$

denotes the set of calls which are combinations of exactly n calls in $\mathcal{C}$.

To decide whether a call graph is good in the sense that it witnesses the termination of all involved functions, we *complete* it, i.e., include all indirect calls, and then check whether all reflexive calls of a function to itself are decreasing. Completeness can be defined by means of combination operation $\circ$:

Definition 3.11 (*Complete Call Graphs*)
A call graph $\mathcal{C}$ is *complete* if $\mathcal{C} \circ \mathcal{C} \subseteq \mathcal{C}$. The *completion* $\mathcal{C}^*$ is the smallest supergraph of $\mathcal{C}$ that is complete.

The completion of a graph $\mathcal{C}$ can be computed by the following saturation algorithm.

Theorem 3.12 (*Completion Algorithm*)
Let $\mathcal{C}$ be a call graph and $(\mathcal{C}_n)_{n\in\mathbb{N}}$ the sequence of call sets defined recursively as follows:

$$\begin{array}{lcl} \mathcal{C}_1 & := & \mathcal{C} \\ \mathcal{C}_{n+1} & := & \mathcal{C}_n \cup (\mathcal{C}_n \circ \mathcal{C}) \end{array}$$

Then there exists an $n \geq 1$ such that $\mathcal{C}_n = \mathcal{C}^*$.

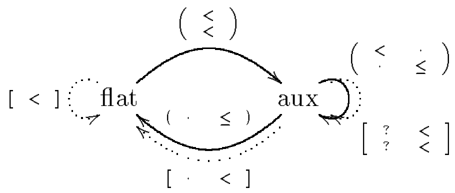
Proof
Since $\mathcal{C}$ has a finite number of nodes and the number of distinct calls between these nodes is finite as well, the $\mathcal{C}_i$ cannot grow endlessly and there is an $n \geq 1$ s.th. $\mathcal{C}_i = \mathcal{C}_n$ for all $i \geq n$. This implies $\mathcal{C}_n \circ \mathcal{C} \subseteq \mathcal{C}_{n+1} \subseteq \mathcal{C}_n$ which iteratively gives us $\mathcal{C}_n \circ \mathcal{C}^i \subseteq \mathcal{C}_n$ for all $i \geq 1$. Because

$$\mathcal{C}_n = \bigcup_{1\leq i\leq n} \mathcal{C}^i,$$

we can infer $\mathcal{C}_n \circ \mathcal{C}_n \subseteq \mathcal{C}_n$. Thus, $\mathcal{C}_n$ is complete, and it remains to show minimality to conclude the proof.

Let $\mathcal{C}'$ be any complete supergraph of $\mathcal{C}$. Then $\mathcal{C} \circ \mathcal{C} \subseteq \mathcal{C}' \circ \mathcal{C}' \subseteq \mathcal{C}'$, and iterating this result, we obtain $\mathcal{C}^i \subseteq \mathcal{C}'$ for all $i \geq 1$. Hence, by the laws of set union, $\mathcal{C}_n \subseteq \mathcal{C}'$ which shows that $\mathcal{C}_n = \mathcal{C}^*$. $\square$

Below we give the completion of our example call graph. Added calls are given by dotted arrows and labelled by call matrices with square brackets.



3.5 Lexicographic Orderings

After completing the call graph, foetus checks whether each reflexive call of a function f to itself is decreasing the size of the arguments. For this, lexicographic orderings are considered and the necessary order of the arguments is computed. In the following we provide a formal description.

Definition 3.13
Given $\mathcal{C}$ a complete call graph and f a function of arity n. We call

$$\mathcal{C}_f := \{\Delta(A) : (f, f, A) \in \mathcal{C}\} \subseteq \mathcal{R}^n$$

the *recursion behavior* of function f. (Δ takes the diagonal of square matrices).

Each tuple in this set represents one possible recursive call of f and how the orders of all parameters are altered in this call. The diagonals of the call matrices are taken because we want to know only how a parameter relates to its old value in the last call to f.

In the following we identify lexicographic orderings on parameters with permutations $\pi \in S_n$ of the arguments. We write $\pi = [k_0 \dots k_{n-1}] \in S_n$ for the permutation $\pi : i \mapsto k_i$. Often not all of the parameters are relevant for termination; these are not listed in the lexicographic ordering and can appear in the permutation in any sequence. We write $\pi = [k_0 \dots k_{m-1}]$ $(m < n)$ for an arbitrary representative of the subset $\{\pi \mid \pi(i) = k_i, i = 0, \dots, m-1\} \subset S_n$.

For the following definition we introduce these abbreviations:

$$\begin{array}{rcll} \pi'_i & = & [k'_0 \dots k'_{i-1} k'_{i+1} \dots k'_{n-1}] \in S_{n-1} & \text{given} \quad \pi \ = \ [k_0 \dots k_{n-1}] \in S_n \\ & & \text{where } k'_j = \left\{ \begin{array}{ll} k_j & \text{if } k_j < k_i \\ k_j - 1 & \text{else} \end{array} \right. & \\ r'_i & = & (k_0, \dots, k_{i-1}, k_{i+1}, \dots, k_{n-1}) \in \mathcal{R}^{n-1} & \text{given} \quad r \ = \ (k_0, \dots, k_{n-1}) \in \mathcal{R}^n \\ B'_i & = & \{r'_i \mid r \in B \wedge r_i \neq <\} \subseteq \mathcal{R}^{n-1} & \text{given} \quad B \ \subseteq \ \mathcal{R}^n \end{array}$$

The tuple r'_i is the result of removing the ith component of r. B'_i is the set of which results from removing all relation tuples containing "<" in the ith position and shortening the remaining tuples by the ith component.

Definition 3.14
Let $B \subseteq \mathcal{R}^n$ be a recursion behavior and $\pi \in S_n$ a permutation. We define the relation "π is a *lexicographic ordering* on B" (π LexOrd B) inductively as follows:

$$\frac{}{\pi \text{ LexOrd } \emptyset}$$

$$\frac{\exists r \in B.\ r_{\pi(0)} = < \quad \forall r \in B.\ r_{\pi(0)} \neq\ ? \quad \pi'_0 \text{ LexOrd } B'_{\pi(0)}}{\pi \text{ LexOrd } B}$$

Note that this definition gives a direct algorithm for computing a lexicographic ordering π for a recursion behavior B: First, find $\pi(0)$ such that $r_{\pi(0)} = <$ for one $r \in B$ but $r_{\pi(0)} \neq\ ?$ for all $r \in B$. Then, continue with π'_0 and $B'_{\pi(0)}$.

Definition 3.15
Let f be a function of arity n in the call graph $\mathcal{C}$. We say $\pi \in S_n$ is a *termination ordering* for f iff π is a lexicographic ordering on the recursion behavior $\mathcal{C}_f$ of f:

$$\pi \text{ TermOrd } f :\Longleftrightarrow \pi \text{ LexOrd } \mathcal{C}_f$$

In our example **`flat`** has recursion behavior $\{(<)\}$ and termination ordering [0] and **`aux`** has recursion behavior $\{(<, \leq), (?, \leq)\}$ and termination ordering [10].

4 An Impredicative Semantic Analysis

Having presented an algorithm to check whether given terms are structurally recursive or not, we want to know whether we can rely on the output of the checker. Thus we have to show that our definition of structural recursion is sound. This involves mutually recursive functions with lexicograhic extensions of the structural ordering, generalized inductive types and higher order functions. Consider the evaluation of an application of a structurally recursive function f:

$$f(u_0) \rightsquigarrow f(u_1) \rightsquigarrow f(u_2) \rightsquigarrow \dots$$

This means that during evaluation of $f(u_0)$ we have to evaluate $f(u_1)$, which again requires evaluation of $f(u_2)$ etc. Since f is structurally recursive, we know that

$$u_0 > u_1 > u_2 > \ldots$$

where ">" is the structural ordering. Surely the evaluation of $f(u_0)$ terminates, if the domain of f is wellfounded, that is, there are no infinite descending chains $u_0 > u_1 > u_2 > \ldots$.

In the following we will prove that in our system all domains are wellfounded. To this end, we first define a semantics for all types (Sect. 4.1). On this semantics we define the structural ordering and show that it is wellfounded (Sect. 4.2). We will extend this ordering lexicographically (Sect. 4.3). Finally we show the soundness of structural recursion by induction over well typed terms (Sect. 4.4).

4.1 Interpretation of the Types

We give a semantics of the types in foetus that captures the "good" values, i.e., those 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. Since we use Knaster-Tarski to define the semantics of μ-types, we must prove monotonicity along its inductive definition.

Definition 4.1 (Semantics)
Given a type $\sigma \in \mathrm{Ty}(\vec{X})$ and closed types $\tau_1, \ldots, \tau_n$ (where $n = |\vec{X}|$) we define the semantics of σ

$$[\![\sigma]\!] : \mathcal{P}\left(\mathrm{Val}^{\tau_1}\right) \times \ldots \times \mathcal{P}\left(\mathrm{Val}^{\tau_n}\right) \to \mathcal{P}\left(\mathrm{Val}^{\sigma(\vec{\tau})}\right)$$

by induction over σ and simultaneously prove monotonicity of $[\![\sigma]\!]$: I.e., $V_i \subseteq W_i$ for $i = 1, \ldots, n$ (where $V_i, W_i \subseteq \mathrm{Val}^{\tau_i}$) implies

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

(Var) $[\![X_i]\!](\vec{V}) := V_i$

$V_i \subseteq W_i$ by assumption.

(Sum) $[\![\Sigma\vec{\sigma}]\!](\vec{V}) := \bigcup_{j=1}^{n}\{\mathrm{in}_j(v) : v \in [\![\sigma_j]\!](\vec{V})\}$

By induction hypothesis $\{\mathrm{in}_j(v) : v \in [\![\sigma_j]\!](\vec{V})\} \subseteq \{\mathrm{in}_j(v) : v \in [\![\sigma_j]\!](\vec{W})\}$ and thus by monotonicity of "$\cup$"

$$[\![\Sigma\vec{\sigma}]\!](\vec{V}) \subseteq [\![\Sigma\vec{\sigma}]\!](\vec{W})$$

(Prod) $[\![\Pi\vec{\sigma}]\!](\vec{V}) := \{(\vec{v}) : v_i \in [\![\sigma_i]\!](\vec{V}) \text{ for } 1 \le i \le n\}$

$[\![\Pi\vec{\sigma}]\!](\vec{V}) \subseteq [\![\Pi\vec{\sigma}]\!](\vec{W})$ by ind.hyp. and monotonicity of the cartesian product.

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

Assume $f \in [\![\sigma \to \tau]\!](\vec{V})$ and $u \in [\![\sigma]\!]$. By definition there is a value $v \in [\![\tau]\!](\vec{V})$ with $f@u \Downarrow v$. By ind.hyp. $v \in [\![\tau]\!](\vec{W})$ and hence $f \in [\![\sigma \to \tau]\!](\vec{W})$.

(Mu) Since we know that $[\![\sigma(\vec{X}, Y)]\!](\vec{V}, -)$ is monontone we can use the theorem of Knaster and Tarski to define $[\![\mu Y.\sigma(\vec{X}, Y)]\!](\vec{V})$ as the smallest set closed under the rule

$$\frac{v \in [\![\sigma]\!](\vec{V}, [\![\mu Y.\sigma]\!](\vec{V}))}{\mathrm{fold}\, v \in [\![\mu Y.\sigma]\!](\vec{V})}$$

For the monotonicity we must prove that $[\![\mu Y.\sigma]\!](\vec{W})$ is closed under this rule: Assume $v \in [\![\sigma]\!](\vec{V}, [\![\mu Y.\sigma]\!](\vec{W}))$. Using the ind.hyp. monotonicity of $[\![\sigma]\!]$ entails $v \in [\![\sigma]\!](\vec{W}, [\![\mu Y.\sigma]\!](\vec{W}))$, hence $\mathrm{fold}\, v \in [\![\mu Y.\sigma]\!](\vec{W})$. □

Proposition 4.2 (Substitution property)

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

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

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

(Mu) "$\subseteq$" We must show that the fixed-point on the right side is closed under the rule defining the one on the left (we omit $\vec{X}$, $\vec{Z}$, $\vec{V}$ and $\vec{W}$):

$$\frac{v \in [\![\sigma(Y, Z)]\!]([\![\tau]\!], [\![\mu Z.\sigma(\tau, Z)]\!])}{\mathrm{fold}\, v \in [\![\mu Z.\sigma(\tau, Z)]\!]}$$

By ind.hyp. we get from the assumption $v \in [\![\sigma(\tau, Z)]\!]([\![\mu Z.\sigma(\tau, Z)]\!])$ which infers the conclusion.
"$\supseteq$" analogously □

Corollary 4.3 (Subset property)
Given $V_i \subseteq [\![\tau_i]\!]$. Then

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

Proof
By iterated application of proposition 4.2 and monotonicity of $[\![\sigma]\!]$. □

4.2 Wellfoundedness of the Structural Ordering on Semantic Values

We now define a transitive structural (pre-)ordering $<$ on semantic values. [5] 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

[5] Note that this is different from the relations $<, \leq$ on terms defined in Sect 3 which formalize the behaviour of the termination checker.

($<$) by the destructor (unfold), whereas case analysis, projection and application keep it on the same level ($\leq$). This is because we need the ordering only 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., $w < v$. Thus we only need to compare values of the same type σ. Since injection (in_j), 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. Hence it is sufficient that unfolding decreases the order strictly.

Definition 4.4 (*Codomain*)
We define the *codomain* of a function $f \in [\![\sigma \to \tau(\vec{X})]\!](\vec{V})$ as

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

Definition 4.5 (*Structural ordering*)
We define a pair of mutually dependent families of relations $<_{\sigma,\tau}, \leq_{\sigma,\tau} \subseteq [\![\sigma]\!] \times [\![\tau]\!]$ inductively as follows:

$$(\leq\mathrm{refl})\ \frac{}{v \leq_{\sigma,\sigma} v} \qquad\qquad (R\mathrm{in})\ \frac{w\ R_{\rho,\sigma_j}\ v}{w\ R_{\rho,\Sigma\vec{\sigma}}\ \mathrm{in}_j(v)}\ R \in \{<, \leq\}$$

$$(R\mathrm{tup})\ \frac{w\ R_{\rho,\sigma_j}\ v_j \quad \text{for some } j \in \{1 \dots n\}}{w\ R_{\rho,\Pi\vec{\sigma}}\ (\vec{v})}\ R \in \{<, \leq\}$$

$$(R\mathrm{arr})\ \frac{w\ R_{\rho,\tau}\ v \quad \text{for some } v \in \mathrm{CoDom}(f)}{w\ R_{\rho,\sigma\to\tau}\ f}\ R \in \{<, \leq\}$$

$$(<\mathrm{fold})\ \frac{w \leq_{\sigma,\tau(\mu X.\tau)}\ v}{w <_{\sigma,\mu X.\tau}\ \mathrm{fold}(v)} \qquad (\leq\mathrm{fold})\ \frac{w \leq_{\sigma,\tau(\mu X.\tau)}\ v}{w \leq_{\sigma,\mu X.\tau}\ \mathrm{fold}(v)}$$

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.

Proposition 4.6 (*Properties*)
The relations $<$ and $\leq$ are transitive, $\leq$ reflexive and $<$ is contained in $\leq$, i.e.,

$$(\leq<) \qquad \frac{w <_{\rho,\tau} v}{w \leq_{\rho,\tau} v}$$

Proof
We show that $w \leq v$ is closed under the rules defining $w < v$: In case of ($<$fold) we must show $w \leq \mathrm{fold}\, v$ from the assumption $w \leq v$: immediately by ($\leq$fold); in all other cases we can apply the respective $\leq$-rule on the induction hypothesis. □

Only after having shown the theorem 4.8 we can verify that our relation is antisymmteric. However, we shall use the term *ordering* in anticipation of this fact. The relation $<$ captures our notion of "smaller values" and allows us to prove wellfoundedness of the computation trees of all values.

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.

Definition 4.7 (*Accessibility*)
Given a family of relations

$$R_{\sigma,\tau} \subseteq [\![\sigma]\!] \times [\![\tau]\!]$$

indexed over closed types $\sigma, \tau \in \mathrm{Ty}$ we define the family of accessible sets $\mathrm{Acc}^\tau_R \subseteq [\![\tau]\!]$ w.r.t. R inductively as follows.

$$(\mathrm{acc}) \qquad \frac{\forall \sigma, w \in [\![\sigma]\!].\ w\ R_{\sigma,\tau}\ v \to w \in \mathrm{Acc}^\sigma_R}{v \in \mathrm{Acc}^\tau_R}$$

Since there is only one introduction rule for Acc^τ_R, we can invert it and obtain the destructor

$$(\mathrm{acc}^{-1}) \qquad \frac{v \in \mathrm{Acc}^\tau_R \quad w\ R_{\sigma,\tau}\ v}{w \in \mathrm{Acc}^\sigma_R}$$

Furthermore the definition yields a wellfounded induction principle: Let P^σ be a family of predicates over values of type σ.

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

Notation. We abbreviate $\mathrm{Acc}^\tau_<$ by Acc^τ. For lists $\vec\rho$ of types we use the abbreviation $\mathrm{Acc}^{\vec\rho} := \mathrm{Acc}^{\rho_1}, \dots, \mathrm{Acc}^{\rho_n}$.

Now we can show that all semantic values are accessible. It would be sufficient to know that all semantic values of closed types $[\![\sigma]\!]$ are accessible. But to prove it for μ-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.

Theorem 4.8 (*All semantic values are accessible*)
Given a type $\sigma \in \mathrm{Ty}(\vec X)$ and a closed type ρ_i for each free variable X_i in σ the following relation holds

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

We prove this theorem by the fact that the generation of semantic values preserves accessibility (see lemma 4.11). But to show this property we need accessibility of less-equal values (lemma 4.10), which again follows from lemma 4.9.

Lemma 4.9 (*Destructors for Acc*)
If a value is accessible, then component values are accessible as well:

$$(\mathrm{accout})\ \frac{\mathrm{in}_j(v) \in \mathrm{Acc}^{\Sigma\vec\sigma}}{v \in \mathrm{Acc}^{\sigma_j}} \qquad (\mathrm{accpi})\ \frac{(\vec v) \in \mathrm{Acc}^{\Pi\vec\sigma}}{v_i \in \mathrm{Acc}^{\sigma_i}}$$

$$(\mathrm{accres})\ \frac{f \in \mathrm{Acc}^{\sigma\to\tau}}{\mathrm{CoDom}(f) \subseteq \mathrm{Acc}^\tau} \qquad (\mathrm{accunf})\ \frac{\mathrm{fold}(v) \in \mathrm{Acc}^{\mu X.\sigma}}{v \in \mathrm{Acc}^{\sigma(\mu X.\sigma)}}$$

Proof

- (accout) Under the assumption (1) $\mathrm{in}_j(v) \in \mathrm{Acc}^{\Sigma\vec{\sigma}}$ we have to show $v \in \mathrm{Acc}^{\sigma_j}$, what by (acc) refines to $w < v \to w \in \mathrm{Acc}^{\rho}$. The assumption (2) $w < v$ entails $w < \mathrm{in}_j(v)$ (by (<in)) and hence using (acc^{-1}) on (1) we obtain $w \in \mathrm{Acc}^{\rho}$.
- (accpi) analogously using (<tup)
- (accres) We have to show $v \in \mathrm{CoDom}(f) \to v \in \mathrm{Acc}^{\tau}$. Assume $v \in \mathrm{CoDom}(f)$ and $w < v$. By (<arr) we get $w < f$ and again (acc^{-1}) proves $w \in \mathrm{Acc}^{\rho}$, hence $v \in \mathrm{Acc}^{\tau}$.
- (accunf) To prove accessibility of v we have to show $w \in \mathrm{Acc}^{\rho}$ under the assumption $w < v$. By ($\leq$<) and (<fold) we get $w < \mathrm{fold}(v)$, and since $\mathrm{fold}(v)$ is accessible by the premise, $w \in \mathrm{Acc}^{\rho}$ by (acc^{-1}). $\square$

Lemma 4.10 (*Accessibility of less-equal values*)
The values less than or equal to an accessible value are accessible themselves.

$$(\mathrm{acc}\leq) \qquad \frac{v \in \mathrm{Acc}^{\tau} \quad w \leq v}{w \in \mathrm{Acc}^{\sigma}}$$

Proof
We show that the relation $w\ R_{\tau,\sigma}\ v :\Longleftrightarrow v \in \mathrm{Acc}^{\sigma} \to w \in \mathrm{Acc}^{\tau}$ is closed under the rules defining $w \leq v$:

- ($\leq$refl) $v \in \mathrm{Acc}^{\sigma} \to v \in \mathrm{Acc}^{\sigma}$ by assumption.
- ($\leq$in) From the ind.hyp. $v \in \mathrm{Acc}^{\sigma_j} \to w \in \mathrm{Acc}^{\rho}$ we have to show

$$\mathrm{in}_j(v) \in \mathrm{Acc}^{\Sigma\vec{\sigma}} \to w \in \mathrm{Acc}^{\rho}$$

  This follows from $\mathrm{in}_j(v) \in \mathrm{Acc}^{\Sigma\vec{\sigma}} \to v \in \mathrm{Acc}^{\sigma}$ (accout).

In the same way we treat the remaining rules ($\leq$pi), ($\leq$arr) and ($\leq$fold), using the propositions (accpi), (accres) and (accunf). $\square$

Lemma 4.11 (*Constructors for Acc*)
A value is accessible if its components are accessible.

$$(\mathrm{accin}) \quad \frac{v \in \mathrm{Acc}^{\sigma_j}}{\mathrm{in}_j(v) \in \mathrm{Acc}^{\Sigma\vec{\sigma}}} \qquad\qquad (\mathrm{acctup}) \quad \frac{v_i \in \mathrm{Acc}^{\sigma_i} \text{ for all } 1 \leq i \leq n}{(\vec{v}) \in \mathrm{Acc}^{\Pi\vec{\sigma}}}$$

$$(\mathrm{accarr}) \quad \frac{f \in [\![\sigma\to\tau]\!] \quad \mathrm{CoDom}(f) \subseteq \mathrm{Acc}^{\tau}}{f \in \mathrm{Acc}^{\sigma\to\tau}} \qquad (\mathrm{accfold}) \quad \frac{v \in \mathrm{Acc}^{\sigma(\mu X.\sigma)}}{\mathrm{fold}(v) \in \mathrm{Acc}^{\mu X.\sigma}}$$

Proof
For all propositions our goal is of the form $[-] \in \mathrm{Acc}^{-}$, what we refine to $w < [-] \to w \in \mathrm{Acc}^{\rho}$. This we prove by case analysis on $w < [-]$.

- (accin) $w < \mathrm{in}_j(v)$ can only be generated by (<in) from $w < v$. By (acc^{-1}) we get $w \in \mathrm{Acc}^{\rho}$.

(acctup) For $w < (\vec{v})$ there must be a $j \in \{1, \dots, n\}$ s.th. $w < v_j$, and since $v_j \in \mathrm{Acc}^{\sigma}_{j}$, $w \in \mathrm{Acc}^{\rho}$ holds.

(accarr) $w < f$ is generated by ($<$arr) from $w < v$ for a $v \in \mathrm{CoDom}(f)$. Since by assumption $v \in \mathrm{Acc}^{\tau}$, we have $w \in \mathrm{Acc}^{\rho}$.

(accfold) For $w < \mathrm{fold}(v)$ the case ($<$fold) matches: $w \leq v$. By (acc$\leq$) we get $w \in \mathrm{Acc}^{\rho}$. $\square$

Proof of Theorem 4.8
We show $[\![\sigma(\vec{X})]\!](\mathrm{Acc}^{\vec{\rho}}) \subseteq \mathrm{Acc}^{\sigma(\vec{\rho})}$ by induction over $\sigma(\vec{X})$.

(Var) $[\![X_j]\!](\mathrm{Acc}^{\vec{\rho}}) = \mathrm{Acc}^{\rho_j} = \mathrm{Acc}^{X_j[\vec{X}:=\vec{\rho}]}$

(Sum) Be $v \in [\![\Sigma\vec{\sigma}]\!](\mathrm{Acc}^{\vec{\rho}})$. Then $v = \mathrm{in}_j(v')$ for a suitable $j \in \{1, \dots, n\}$ and $v' \in [\![\sigma_j]\!](\mathrm{Acc}^{\vec{\rho}})$. By ind.hyp. we have $v' \in \mathrm{Acc}^{\sigma_j(\vec{\rho})}$, hence by (accin) $v \in \mathrm{Acc}^{\Sigma\vec{\sigma}(\vec{\rho})}$.

(Prod) Be $(\vec{v}) \in [\![\Pi\vec{\sigma}]\!](\mathrm{Acc}^{\vec{\rho}})$. Since $v_i \in [\![\sigma_i]\!](\mathrm{Acc}^{\vec{\rho}})$ for all $1 \leq i \leq n$, by ind.hyp. $v_i \in \mathrm{Acc}^{\sigma_i(\vec{\rho})}$, hence by (acctup) $(\vec{v}) \in \mathrm{Acc}^{\Pi\vec{\sigma}(\vec{\rho})}$.

(Arr) We assume $f \in [\![\sigma \to \tau(\vec{X})]\!](\mathrm{Acc}^{\vec{\rho}})$. Since $\mathrm{CoDom}(f) \subseteq [\![\tau(\vec{X})]\!](\mathrm{Acc}^{\vec{\rho}})$ by definition, the ind.hyp. entails $\mathrm{CoDom}(f) \subseteq \mathrm{Acc}^{\tau(\vec{\rho})}$. Corollary 4.3 infers $f \in [\![\sigma \to \tau(\vec{\rho})]\!]$ and hence by (accarr) $f \in \mathrm{Acc}^{\sigma\to\tau(\vec{\rho})}$.

(Mu) We prove that $\mathrm{Acc}^{\mu Y.\sigma(\vec{\rho},Y)}$ is closed under the rule that is defining $[\![\mu Y.\sigma(\vec{X}, Y)]\!](\mathrm{Acc}^{\vec{\rho}})$, i.e., we show

$$\frac{v \in [\![\sigma(\vec{X}, Y)]\!](\mathrm{Acc}^{\vec{\rho}}, \mathrm{Acc}^{\mu Y.\sigma(\vec{\rho},Y)})}{\mathrm{fold}\, v \in \mathrm{Acc}^{\mu Y.\sigma(\vec{\rho},Y)}}$$

Applying the ind.hyp. on the premise entails $v \in \mathrm{Acc}^{\sigma(\vec{\rho},\mu Y.\sigma(\vec{\rho},Y))}$, and (accfold) infers the conclusion. $\square$

Corollary 4.12
The semantic values coincide with the accessible ones.

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

Proof
"$\subseteq$" by the theorem, "$\supseteq$": Since for closed types ρ from the lemma we immediately get $[\![\rho]\!] = \mathrm{Acc}^{\rho}$, we have by proposition 4.2 $\mathrm{Acc}^{\sigma(\vec{\rho})} = [\![\sigma(\vec{\rho})]\!] = [\![\sigma(\vec{X})]\!]([\![\vec{\rho}]\!]) = [\![\sigma(\vec{X})]\!](\mathrm{Acc}^{\vec{\rho}})$. $\square$

4.3 Wellfoundedness of the Lexicographic Extension

Since the termination checker allows descent over lexicographic orderings in recursive functions, we have to define it on values and prove that it is wellfounded as well. When denoting tuples in vector notation, we allow ourselves to omit the enclosing parentheses, i.e., for $(\vec{v}) \in [\![\Pi\vec{\sigma}]\!]$ we write just $\vec{v}$. Furthermore we use the abbreviations for shortened vectors and permutations as defined in Sect. 3.

Definition 4.13 (*Lexicographic Ordering*)
Given closed types $\vec{\sigma} = \sigma_1, \dots, \sigma_n$ and a permutation $\pi \in S_n$ we inductively define $\prec^{\pi}_{\Pi\vec{\sigma}} \subseteq [\![\Pi\vec{\sigma}]\!] \times [\![\Pi\vec{\sigma}]\!]$ as follows:

$$(\text{lex}<)\ \frac{v_{\pi(0)} < w_{\pi(0)}}{\vec{v} \prec^{\pi}_{\Pi\vec{\sigma}} \vec{w}} \quad (\text{lex}\leq)\ \frac{v_{\pi(0)} \leq w_{\pi(0)} \quad \vec{v}'_{\pi(0)} \prec^{\pi'_0}_{\Pi\vec{\sigma}'_{\pi(0)}} \vec{w}'_{\pi(0)}}{\vec{v} \prec^{\pi}_{\Pi\vec{\sigma}} \vec{w}}$$

Proposition 4.14 (*Wellfoundedness of the Lexicographic Ordering*)
Given closed types $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ the following relation holds:

$$[\![\Pi\vec{\sigma}]\!] \subseteq \mathrm{Acc}^{\Pi\vec{\sigma}}_{\prec}$$

To show this proposition we need the following lemma:

Lemma 4.15

$$\frac{\vec{v} \in \mathrm{Acc}_{\prec} \quad w_j \leq v_j \quad w_i = v_i \text{ for all } i \neq j}{\vec{w} \in \mathrm{Acc}_{\prec}}$$

Proof
By case analysis and transitivity of the structural ordering show $\vec{u} \prec \vec{w} \rightarrow \vec{u} \prec \vec{v}$ for all $\vec{u}$. □

Proof of Prop. 4.14
By induction on n. We make extensive use of the wellfounded induction principle.

$n = 0$ We have to show $() \in \mathrm{Acc}^{1}_{\prec}$, what refines to

$$\forall \vec{w} \prec ().\ \vec{w} \in \mathrm{Acc}_{\prec}$$

This is true since no case matches for $\vec{w} \prec ()$.

$n \rightarrow n+1$ Or goal is

$$v_{\pi(0)} \in [\![\sigma_{\pi(0)}]\!] \rightarrow \vec{v}'_{\pi(0)} \in [\![\vec{\sigma}'_{\pi(0)}]\!] \rightarrow \vec{v} \in \mathrm{Acc}^{\Pi\vec{\sigma}}_{\prec^{\pi}}$$

Using theorem 4.8 we can replace $[\![\sigma_{\pi(0)}]\!]$ by $\mathrm{Acc}^{\sigma_{\pi(0)}}$ and apply well-founded induction on it. This gives us the hypothesis

$$\forall \vec{w} \in [\![\Pi\sigma]\!].w_{\pi(0)} < v_{\pi(0)} \rightarrow \vec{w} \in \mathrm{Acc}_{\prec} \tag{5}$$

to show $\vec{v}'_{\pi(0)} \in [\![\vec{\sigma}'_{\pi(0)}]\!] \rightarrow \vec{v} \in \mathrm{Acc}_{\prec}$. Using the ind.hyp. we can replace $[\![\vec{\sigma}'_{\pi(0)}]\!]$ by $\mathrm{Acc}^{\Pi\vec{\sigma}'_{\pi(0)}}_{\prec}$ and apply well-founded induction, obtaining

$$\forall \vec{w} \in [\![\Pi\sigma]\!].w_{\pi(0)} = v_{\pi(0)} \rightarrow \vec{w}'_{\pi(0)} \prec \vec{v}'_{\pi(0)} \rightarrow \vec{w} \in \mathrm{Acc}_{\prec} \tag{6}$$

to show $\vec{v} \in \mathrm{Acc}_{\prec}$. For this we have to prove $\vec{w} \prec \vec{v} \rightarrow \vec{w} \in \mathrm{Acc}_{\prec}$. Case analysis on the generation of $\vec{w} \prec \vec{v}$:
Case $w_{\pi(0)} < v_{\pi(0)}$ (lex<): According to (5) $\vec{w} \in \mathrm{Acc}_{\prec}$.
Case $w_{\pi(0)} \leq v_{\pi(0)}, \vec{w}'_{\pi(0)} \prec \vec{v}'_{\pi(0)}$ (lex≤): Hypothesis (6) entails $\vec{u} \in \mathrm{Acc}_{\prec}$, where $u_{\pi(0)} = v_{\pi(0)}$ and $u_i = w_i$ for $i \neq \pi(0)$. Following lemma 4.15 $\vec{w} \in \mathrm{Acc}_{\prec}$. □

Corollary 4.16
With regard to the lexicographic ordering, the good value tuples are exactly the accessible ones.

$$[\![\Pi\vec{\sigma}]\!] = \mathrm{Acc}^{\Pi\vec{\sigma}}_{\prec}$$

4.4 Soundness of Structural Recursion

To show that our system is sound, we will prove normalization where we restrict recursive terms to structurally recursive ones. Restricting environments to good ones, i.e., to those containing only semantic values, we show that each term normalizes with regard to the operational semantics defined in Sect. 2.

To this end, we will define good terms TM, good environments $[\![\Gamma]\!]$ and good closures. After stating that structurally recursive functions terminate, we can do the normalization proof mechanically.

Definition 4.17 (*Good Environments*)
The subset of *environments of semantic values* over context $\Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ is naturally defined as

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

Definition 4.18
We say a closure $c \in \mathrm{Cl}^\sigma$ *terminates* $c \Downarrow$ iff it evaluates to some value $v \in [\![\sigma]\!]$:

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

Definition 4.19 (*Structural Recursiveness*)
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 $w \prec v$:

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

Proposition 4.20 (*Structurally recursive functions terminate*)
Let $\mathrm{rec}\, g.\, t \in \mathrm{SR}^{\sigma\to\tau}$ and $e \in [\![\Gamma]\!]$. Then

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

Proof
Assuming $e \in [\![\Gamma]\!]$ we expand the first premise to

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

Applying wellfounded induction with the familiy of predicates $P^\rho \subseteq [\![\rho]\!]$ $(\rho \in \mathrm{Ty})$ defined as

$$P^\rho(v) := \begin{cases} \langle \mathrm{rec}\, g.\, t; e\rangle @ v \Downarrow & \text{if } \rho = \sigma \\ \text{true} & \text{else} \end{cases}$$

yields $\forall v \in \mathrm{Acc}^\sigma_\prec.\, \langle \mathrm{rec}\, g.\, t; e\rangle @ v \Downarrow$, which is equivalent to our claim since all values are accessible ($\mathrm{Acc}^\sigma_\prec = [\![\sigma]\!]$, see corollary 4.16). □

With $\mathrm{SR}^{\sigma\to\tau}$ we have given a *semantical* criterion for structural recursion. This can be replaced by *syntactical* criteria for certain classes of structurally recursive functions. In (Abel, 2000), we have provided a criterion that captures non-mutually recursive functions which terminate by lexicographic extensions of the structural ordering. Of course, we were required to give a proof that all such syntactically structurally recursive functions are also semantically structurally recursive.

Definition 4.21 (*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 Def. 2.2), we only change Tm to TM and label the rule in CAPITAL letters, e.g.

$$\text{(TUP)} \qquad \frac{t_i \in \mathrm{TM}_i^{\sigma}[\Gamma] \text{ for } 1 \leq i \leq n}{(t_1, \ldots, t_n) \in \mathrm{TM}^{\Pi\vec{\sigma}}[\Gamma]}$$

Only the rule (rec) is replaced by

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

Definition 4.22 (*Good closures*)
Consequently the set of good closures CL^{τ} of type τ is defined as

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

Again $\mathrm{CL}^{\tau} \subseteq \mathrm{Cl}^{\tau}$, hence we can use our operational semantics $\Downarrow$ on good closures as well. Now we are ready to show normalization.

Theorem 4.23 (*Normalization*)
Let $t \in \mathrm{TM}^{\sigma}[\Gamma]$ be a good term of type σ over context Γ and $e \in [\![\Gamma]\!]$ a good environment. Then there exists a $v \in [\![\sigma]\!]$ such that

$$\langle t; e\rangle \Downarrow v.$$

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

(VAR) Since $e^{\Gamma}, x^{\sigma} = v, d^{\Gamma'} \in [\![\Gamma, x^{\sigma}, \Gamma']\!]$ by (opvar) we prove

$$\langle x[\Gamma, x^{\sigma}\Gamma']; e, x = v, d\rangle \Downarrow v \in [\![\sigma]\!]$$

(IN) By ind.hyp. $\langle t^{\sigma_j}; e\rangle \Downarrow v \in [\![\sigma_j]\!]$, thus by (opin)

$$\langle \mathrm{in}_j(t)^{\Sigma\vec{\sigma}}; e\rangle \Downarrow \mathrm{in}_j(v) \in [\![\Sigma\vec{\sigma}]\!]$$

(CASE) We must show

$$\langle \mathrm{case}(u^{\Sigma\vec{\sigma}}[\Gamma], x_1.\, t_1^{\rho}[\Gamma, x_1^{\sigma_1}], \ldots, x_n.\, t_n^{\rho}[\Gamma, x_n^{\sigma_n}]); e\rangle \Downarrow v \in [\![\rho]\!]$$

By ind.hyp. we have $\langle u^{\Sigma\vec{\sigma}}[\Gamma]; e\rangle \Downarrow w' \in [\![\Sigma\vec{\sigma}]\!]$. Now w' must be of form $\mathrm{in}_j(w^{\sigma_j})$: By ind.hyp. we get $\langle t_j^{\rho}[\Gamma, x_j^{\sigma_j}]; e, x_j = w\rangle \Downarrow v \in [\![\rho]\!]$, hence by (opcase) we prove our claim.

(TUP) Here we show

$$\langle (t_1^{\sigma_1}, \ldots, t_n^{\sigma_n}); e\rangle \Downarrow$$

By ind.hyp. we have $\langle t_i; e\rangle \Downarrow v_i \in [\![\sigma_i]\!]$ for all $1 \leq i \leq n$ hence by (optup) $\langle(\vec{t}); e\rangle \Downarrow (\vec{v})$ which is in $[\![\Pi\vec{\sigma}]\!]$ by definition.

(PI) By ind.hyp. $\langle t^{\Pi\vec{\sigma}}; e\rangle \Downarrow (\vec{v}) \in [\![\Pi\vec{\sigma}]\!]$, hence by (oppi)

$$\langle \pi_j(t); e\rangle \Downarrow v_j \in [\![\sigma]\!]$$

for any $1 \leq j \leq n$.

(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 4.20 since $\mathrm{rec}\, g.\, t \in \mathrm{SR}^{\sigma\to\tau}$ by definition.

(APP) By ind.hyp. 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 futhermore by (opapp)

$$\langle t\, s; e\rangle \Downarrow$$

(FOLD) By ind.hyp. $\langle t; e\rangle \Downarrow v \in [\![\sigma(\mu X.\sigma)]\!]$, hence by (opfold)

$$\langle \mathrm{fold}(t); e\rangle \Downarrow \mathrm{fold}(v)$$

which is in $[\![\mu X.\sigma]\!]$ by definition.

(UNFOLD) By ind.hyp. $\langle t; e\rangle \Downarrow \mathrm{fold}(v) \in [\![\mu X.\sigma]\!]$, hence

$$\langle \mathrm{unfold}(t); e\rangle \Downarrow v$$

by (opunfold), which is in $[\![\sigma(\mu X.\sigma)]\!]$. □

Note that we have also shown normalization for mutual recursive terms (as for instance our example **flat**): They are handled by case (REC). Since in all terms accepted by termination checker all recursive subterms are in SR, they are "good" by Proposition 4.20.

5 A Predicative Analysis using Set Based Operators

In definition 4.1 we used the theorem of Knaster-Tarski to construct the interpretation of μ-Types. In this section we shall show that impredicative reasoning is not necessary and can be replaced by strictly positive inductive definitions.

5.1 On Inductive Definitions

A set $\mu \subseteq U$ is considered to be inductively defined if it is the fixpoint of some operator $\Phi : \mathcal{P}(U) \to \mathcal{P}(U)$. Elements of μ can be constructed and used according to the following introduction and elimination rules:

$$(\mathrm{intro})\,\frac{v \in \Phi(\mu)}{v \in \mu} \qquad (\mathrm{elim})\,\frac{v \in \mu \quad \Phi(V) \subseteq V}{v \in V}$$

However, we have to impose a side condition on Φ to ensure that the fixpoint exists. Using impredicative reasoning, it is suffient that Φ is *monotone*. Then by the theorem of Knaster and Tarski (1955) the fixpoint exists as

$$\mu := \bigcap\{V \subseteq U : \Phi(V) \subseteq V\}$$

An example for a monotone operator, whose fixpoint is the set $[\![\mu X.\sigma]\!]$, has been given in Sect. 4.1:

$$\Phi_{[\![\mu X.\sigma]\!]}(V) := \{\mathrm{fold}\, v : v \in [\![\sigma]\!](V)\}$$

The operator $\Phi_{[\![\mu X.\sigma]\!]}$ is monotone since $[\![\sigma]\!]$ is monotone, which we could prove in Def. 4.1 because X appeares strictly positive in σ.

However, in a predicative theory v is an element of the fixpoint μ if and only if there exists a well-founded deriviation tree of the proposition $v \in \mu$ using the rules of the theory including (intro) and (elim). This tree is always well-founded if the operator Φ is *strictly positive*. An example is the operator defining accessible sets given in Sect. 4.2 (simplified for clarity):

$$\Phi_{\mathrm{Acc}^\rho}(V) := \{v \in [\![\rho]\!] : \forall w < v.\ w \in V\}$$

The fixpoint of Φ_{Acc^ρ} is the set Acc^ρ of accessible values of type ρ. The variable V appears never on the right hand side of an arrow and thus the operator is strictly positive. In constrast, the operator $\Phi_{[\![\mu X.\sigma]\!]}$ given above is not strictly positive since the variable V appears as an argument of $[\![\sigma]\!]$. If $[\![\sigma]\!](V)$ was just a notational definition, we could expand it and check whether V appears strictly positive, but unfortunately the interpretation $[\![-]\!]$ is defined by recursion on the structure of the type σ and thus we cannot know what $[\![\sigma]\!]$ "does" with its argument.

A solution to the problem could be to replace $\Phi_{[\![\mu X.\sigma]\!]}$ by a semantically equivalent operator which is strictly positive. Indeed, that can be done using Peter Aczel's set based operators, as we show in the next section. An alternative would be to reformulate the definition of types s.th. the interpretation can be given by a strictly positive inductive definition directly. This approach has been carried out by Holger Benl (1998) using vector notation and simulatenously inductive types instead of interleaving inductive types. However, his proof could only be implemented in a predicative theorem proving system that considers vector notation as primitive.

5.2 Interpreting Strictly Positive Types

The interpretation $[\![\sigma]\!]$ of a strictly positive inductive type σ is monotone, but this does not seem to capture the fundamental property of strictly positive types. Indeed, we could show in (Abel, 1999) that a wider class of types enjoy this property: the positive types. In the following we will show that strictly positive types are interpreted as operators that are *set based*, which makes their definition by a predicative inductive definition possible. We consider set-basedness to be the characterizing property of stricly positive types.

We extend Def. 4.1 of the semantics by the requirement that each operator has to come with an *urelement* relation $\mathcal{U}$. The construction follows closely the one presented in (Abel & Altenkirch, 2000). Here the verification is simpler because we do not need closure rules to achieve saturatedness.

Definition 5.1 (Semantics)
For $\sigma \in \mathrm{Ty}(\vec{X})$ and closed types $\tau_1, \ldots, \tau_n$ we define

$$[\![\sigma]\!] : \mathcal{P}\left(\mathrm{Val}^{\tau_1}\right) \times \ldots \times \mathcal{P}\left(\mathrm{Val}^{\tau_n}\right) \to \mathcal{P}\left(\mathrm{Val}^{\sigma(\vec{\tau})}\right)$$

as in definition 4.1 but we additionally require that there are relations

$$\mathcal{U}_i^\sigma \subseteq \mathrm{Val}^{\tau_i} \times \mathrm{Val}^{\sigma(\vec{\tau})} \qquad \text{for } 1 \leq i \leq n$$

such that for all $1 \leq i \leq n$, $v \in \mathrm{Val}^{\sigma(\vec{\tau})}$ and $u \in \mathrm{Val}^{\tau_i}$:

$$\frac{v \in [\![\sigma]\!](\vec{V}) \qquad u\,\mathcal{U}_i^\sigma\ v}{u \in V_i}\,(\mathrm{sb1}) \qquad \frac{v \in [\![\sigma]\!](\mathrm{Val}^{\vec{\tau}})}{v \in [\![\sigma]\!](\vec{\mathcal{U}}^\sigma(v))}\,(\mathrm{sb2})$$

The definition of $[\![\sigma]\!]$ in the cases (Var), (Sum), (Prod), (Arr) is the same as before. The interpretation of μ-types is replaced by

$$\frac{v \in [\![\sigma]\!](\vec{V}, \mathrm{Val}^{\mu X.\sigma}) \qquad \forall u.\ u\,\mathcal{U}_{n+1}^\sigma\ v \to u \in [\![\mu X.\sigma]\!](\vec{V})}{\mathrm{fold}(v) \in [\![\mu X.\sigma]\!](\vec{V})}$$

Below we will define the relations $\mathcal{U}_i^\sigma$ and show that they satisfy the required properties.

The rule (sb1), which can be read as $v \in [\![\sigma]\!](\vec{V}) \to \mathcal{U}_i^\sigma(v) \subseteq V_i$, states that the relations $\mathcal{U}_i$ "sort" the urelements of v back into the sets V_i they came from. More precisely, if $u\,\mathcal{U}_i\ v$ holds then u is a urelement of v that came from value set V_i. Rule (sb2) expresses that a value v can be reconstructed from its urelements.

Let us demonstrate the urelement-relation in the case of lists, where it actually implements an element relation. Lists of element type X can be defined as

$$\mathbf{List}(X) \equiv \mu Y.\, 1 + X \times Y$$

Given a closed type τ and a set of values $V \subseteq \mathrm{Val}^\tau$ lists can be introduced by the following two rules:

$$\frac{}{[] \in [\![\mathbf{List}]\!](V)} \qquad \frac{u \in V \qquad v \in [\![\mathbf{List}]\!](V)}{u :: v \in [\![\mathbf{List}]\!](V)}$$

For readability we use the notational definitions $[] \equiv \mathrm{fold}(\mathrm{in}_1())$ and $u :: v \equiv \mathrm{fold}(\mathrm{in}_2(u, v))$. The urelement relation can be defined by these rules:

$$\frac{}{u\,\mathcal{U}^{\mathbf{List}}\ u :: v} \qquad \frac{u\,\mathcal{U}^{\mathbf{List}}\ v}{u\,\mathcal{U}^{\mathbf{List}}\ w :: v}$$

Property (sb1) for $\mathcal{U}^{\mathbf{List}}$ states that for all lists v all elements $u\,\mathcal{U}\ v$ must have come from V. Property (sb2), which states that a list can be constructed from its elements, is non-trivial.

Proposition 5.2 (set basedness for lists)

(sb1) If $v \in [\![\mathbf{List}]\!](V)$ and $u\,\mathcal{U}\,v$ then $u \in V$.
(sb2) If $v \in [\![\mathbf{List}(\tau)]\!]$ then $v \in [\![\mathbf{List}]\!](\mathcal{U}(v))$.

Proof
(sb1) By induction on $u\,\mathcal{U}\,v$.
(sb2) By induction on $v \in [\![\mathbf{List}(\tau)]\!]$: The base case is trivial. For $v = u :: v'$ we have by induction hypothesis that $v' \in [\![\mathbf{List}]\!](\mathcal{U}(v'))$. By the definiton of $\mathcal{U}$ for lists it holds that $\mathcal{U}(v') \subseteq \mathcal{U}(u :: v')$. Thus we can make use of the monotonicity of $[\![\mathbf{List}]\!]$ and derive $v' \in [\![\mathbf{List}]\!](\mathcal{U}(u :: v'))$ from which our goal follows. $\square$

These proofs serve as models for the proofs in the general inductive case.

The following theorem illustrates the benefit of set basedness: The interpretation of strictly positive types can be made strictly positive itself.

Theorem 5.3 (*Fundamental property of strictly positive types*)
Every $[\![\sigma]\!](\vec{V})$ is equivalent to a predicate which is strictly positive in $\vec{V}$ - that is for $v \in \mathrm{Val}^{\sigma(\vec{\tau})}$:

$$v \in [\![\sigma]\!](\vec{V}) \quad \Longleftrightarrow \quad v \in [\![\sigma]\!](\mathrm{Val}^{\vec{\tau}}) \ \wedge \ \forall i.\, \mathcal{U}_i^{\sigma}(v) \subseteq V_i$$

Proof
$\Rightarrow$ Assuming $v \in [\![\sigma]\!](\vec{V})$, we obtain $v \in [\![\sigma]\!](\mathrm{Val}^{\vec{\tau}})$ by monotonicity and $\mathcal{U}_i^{\sigma}(v) \subseteq V_i$ for all i by (sb1).
$\Leftarrow$ Using (sb2), $v \in [\![\sigma]\!](\mathrm{Val}^{\vec{\tau}})$ entails

$$v \in [\![\sigma]\!](\vec{\mathcal{U}}^{\sigma}(v))$$

Since by assumption $\vec{\mathcal{U}}^{\sigma}(v) \subseteq \vec{V}$ (component wise), we can derive $v \in [\![\sigma]\!](\vec{V})$ by monotonicity. $\square$

Having proven this fundamental property, we see at the first glance that the new, strictly positive operator for the interpretation of inductive types given in Def. 5.1 constructs the same semantics as the old in Def. 4.1, which can be written as

$$\frac{v \in [\![\sigma]\!](\vec{V}, [\![\mu X.\sigma]\!](\vec{V}))}{\mathrm{fold}(v) \in [\![\mu X.\sigma]\!](\vec{V})}$$

The premises of the old rule match the left hand side of the equivalence in Theorem 5.3, and the premises of the new rule the right hand side.

We now complete Def. 5.1 and give the definition of the urelement relation $\mathcal{U}^{\sigma}$ by induction over the type σ. In each case it is defined inductively by a set of strictly positive rules and we have to prove the laws of set basedness. In each case (sb1) is proven by induction over the derivation of $u\ \mathcal{U}\ v$ and (sb2) is shown by induction over $v \in [\![\sigma]\!](\mathrm{Val}^{\vec{\tau}})$, using monotonicity of the operators and the set basedness properties for the already defined types. We only give a detailed proof for the non-tricvial case of μ-types.

(Var)

$$\frac{v \in V_i}{v\,\mathcal{U}_i^{X_i}\ v}$$

(Sum) For all $1 \leq j \leq |\vec{\sigma}|$

$$\frac{u\,\mathcal{U}_i^{\sigma_j}\ v}{u\,\mathcal{U}_i^{\Sigma\vec{\sigma}}\ \mathrm{in}_j(v)}$$

(Prod)

$$\frac{u\,\mathcal{U}_i^{\sigma_j}\ v_j \quad \text{for all } 1 \leq j \leq |\vec{\sigma}|}{u\,\mathcal{U}_i^{\Pi\vec{\sigma}}\ (\vec{v})}$$

(Arr)

$$\frac{v \in \mathrm{CoDom}(f) \qquad u\,\mathcal{U}_i^{\tau}\ v}{u\,\mathcal{U}_i^{\sigma\to\tau}\ f}$$

(Mu) $\mathcal{U}^{\mu X.\sigma}$ is defined inductively by the following two rules:

$$(\text{non-rec})\ \frac{1 \leq i \leq n \quad u\,\mathcal{U}_i^{\sigma}\ v}{u\,\mathcal{U}_i^{\mu X.\sigma}\ \mathrm{fold}(v)} \qquad (\text{rec})\ \frac{v'\ \mathcal{U}_{n+1}^{\sigma}\ v \quad u\,\mathcal{U}_i^{\mu X.\sigma}\ v'}{u\,\mathcal{U}_i^{\mu X.\sigma}\ \mathrm{fold}(v)}$$

(sb1) We show that

$$u\ R_i\ v :\Longleftrightarrow v \in [\![\mu X.\sigma]\!](\vec{V}) \to u \in V_i$$

is closed under the rules defining $\mathcal{U}_i^{\mu X.\sigma}$

non-rec If $\mathrm{fold}(v) \in [\![\mu X.\sigma]\!](\vec{V})$ then $v \in [\![\sigma]\!](\vec{V}, [\![\mu X.\sigma]\!](\vec{V}))$ and hence (sb1) for σ entails that $u\,\mathcal{U}_i^{\sigma}\ v$ implies $u \in V_i$.

rec As before we have $v \in [\![\sigma]\!](\vec{V}, [\![\mu X.\sigma]\!](\vec{V}))$. Hence, using (sb1) for σ, the first premise $v'\ \mathcal{U}_{n+1}^{\sigma}\ v$ implies $v' \in [\![\mu X.\sigma]\!](\vec{V})$. Now we use the second premise $u\ R_i\ v'$ to conclude $u \in V_i$.

(sb2) We show that the set

$$Q = \{v \mid v \in [\![\mu X.\sigma]\!](\vec{\mathcal{U}}^{\mu X.\sigma}(v))\}$$

is closed under the rule defining $[\![\mu X.\sigma]\!](\mathrm{Val}^{\vec{\tau}})$. We assume

$$v \in [\![\sigma]\!](\mathrm{Val}^{\vec{\tau}}, Q) \tag{7}$$

which by using (sb2) for σ entails

$$v \in [\![\sigma]\!](\vec{\mathcal{U}}^{\sigma}(v)) \tag{8}$$

To show that $\mathrm{fold}(v) \in Q$ it suffices to show

$$v \in [\![\sigma]\!](\vec{\mathcal{U}}^{\mu X.\sigma}(\mathrm{fold}(v)), [\![\mu X.\sigma]\!](\vec{\mathcal{U}}^{\mu X.\sigma}(\mathrm{fold}(v))))$$

We derive this from 8 using (mon), which leaves us two subgoals

1. We have to show that $\mathcal{U}_i^{\sigma}(v) \subseteq \mathcal{U}_i^{\mu X.\sigma}(\mathrm{fold}(v))$ which follows from (non-rec).
2. To show $\mathcal{U}_{n+1}^{\sigma}(v) \subseteq [\![\mu X.\sigma]\!](\vec{\mathcal{U}}^{\mu X.\sigma}(\mathrm{fold}(v)))$ assume

$$v'\ \mathcal{U}_{n+1}^{\sigma}\ v \tag{9}$$

Under this assumption we have that $\mathcal{U}_i^{\mu X.\sigma}(v') \subseteq \mathcal{U}_i^{\mu X.\sigma}(\mathrm{fold}(v))$ by (rec). Using (sb1) for σ on 7 and 9 we have that $v' \in Q$, i.e.,

$$v' \in [\![\mu X.\sigma]\!](\vec{\mathcal{U}}^{\mu X.\sigma}(v'))$$

and hence using (mon) $v' \in [\![\mu X.\sigma]\!](\vec{\mathcal{U}}^{\mu X.\sigma}(\mathrm{fold}(v)))$. $\square$

6 Conclusions and Further Work

We have introduced a termination checker foetus for structurally recursive terms that recognizes also descent via a lexicographic ordering. We have given an interpretation of the types, transfered the ordering onto the interpretation and thus have shown that (even mutually) recursive terms accepted by foetus are sound.

In (Abel, 2000) our work is continued: We introduce a judgement that implements a *syntactic* check whether functions are structurally recursive. This judgement is an abstract description of the algorithm that foetus uses to decide whether functions are structurally recursive or not. We then prove that this syntactic check is sound w.r.t. to the semantics given here. This proof is restricted to non-mutual recursion, the proof for mutual recursion is in preparation.

We would like to extend our work to (predicatively) polymorphic, coinductive and dependent types. To handle corecursive functions especially the interplay between guarded and structurally recursive has to be considered. Introducing dependent types, more effort has to be done to check the totality of a pattern.

6.1 Related Work

Publications on termination are numerous since research in this area has been carried out since the very beginnings of computer science. Much work has been done in the context of term rewriting systems and we want to cite the seminal investigations of Christoph Walther (1988):

Walther has described *reduction checking*, which is sometimes referred to as “Walther recursion” (McAllester & Arkoudas, 1996). His *estimation calculus* examines whether functions are terminating and 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. More work on the estimation calculus has been done by Bundy and others (Gow *et al.*, 1999).

Closely related to our work, though in the area of extensible term rewriting systems is the work of Blanqui, Jouannaud and Okada (2001). They define structurally recursive terms syntactically using side conditions by their *Extended General Schema*. Since they do not handle mutually recursive functions, their approach seems to have the same strength as the formal system we present in Sect. 3.2.

Gimenéz (1995) presents a syntactic translation of structural (and guarded) recursive definitions into (dependently typed) recursors in the context of the *Coq* system (Barras *et al.* , 2000)—based on the Calculus of Constructions (Coquand &

Huet, 1988). We avoid having to introduce and analyze a type theory with recursors but consider structural recursion as a primitive principle. The consideration of predicativity does not play a role in Giminéz' work because *Coq* is impredicative anyway.

More recently Gimenéz (1998) has shown how to integrate guardedness and termination check into type-checking. He uses a tagged version of the types to indicate whether its inhabitants are guarded by a constructor. This technique yields partial reduction checking for free: The type system can capture whether a function is a preserver, but not whether it is a reducer. His type system does not contain Σ-types and thus he cannot handle lexicographic descent. An extension to Σ-types is non-trivial since it is not obvious how to incorporate lexicographic products into his subtyping calculus.

Gimenéz motivated his new technique as follows: Side-conditions in typing rules that ensure termination are undesirable since it is difficult to construct a semantics for the types, which is needed in normalization proofs. In this article we manage to construct a sound interpretation for the types and a rigid normalization proof is given by one of the authors (Abel, 2000).

Amadio and Coupet-Grimal (1998) analyse a λ-calculus with *coinductive* types where the guardedness check is woven into the typing rules. They give a PER model of the types and interpret the terms in the untyped λ-calculus (in contrast to our *observable values* interpretation). Furthermore they give a proof of strong normalization for a restricted reduction rule for corecursively constructed data. Codata are only unfolded in the head position of a case-construct, which corresponds to our approach to expand recursive functions only in case of an application.

Telford and Turner (1999) are investigating *Elementary Strong Functional Programming*, i.e., functional programming languages where only terminating functions can be defined. Technically they use *abstract interpretations* to ensure termination. They can handle a wider class of functions since they keep track not *whether* an argument is decreasing but *how much* it is decreasing or increasing, thus allowing temporary growth that is compensated by sufficient shinkage later. We consider their technique as a promising alternative to our logical approach. However the focus of our work is not maximal completeness but a solid theoretical foundation.

Lee, Jones and Ben-Amram (2001) introduce a new paradigm into reasoning about termination: "a program terminates on all inputs if every *infinite* call sequence (...) would cause an infinite descent in some data values". Thus he characterizes terminating functions as those accepted by certain Büchi automata. For the practical implementation of termination checking he proposes an algorithm based on *size-change graphs*, a notion different from our call graphs but related to them. Its follows a principle similar to our call graph completion, thus their complexity results should apply to our algorithm as well.

Finally, Pientka and Pfenning (2000) have implemented reduction and termination checking for the logical framework LF (Harper *et al.*, 1993). Their algorithm does not *infer* termination orderings, but *checks* termination using an ordering the user specifies. The formulation of the algorithm is judgment-based like ours and justified by a cut admissibility argument rather than a soundness proof as in our case.

6.2 Acknowledgements

The authors like to thank Catarina and Thierry Coquand for inspiring e-mail exchange and discussions. The predicative semantics in Section 5 was inspired by Thierry Coquand's proposal that this should be carried out using Peter Aczel's set based operators. We thank Jeremy Avigad, Ralph Matthes, Frank Pfenning and the anonymous referee for comments on draft versions of this paper.

The first author thanks the creator of the universe who gave mankind the gift of creativity and abstract thinking.

References

Abel, Andreas. (1999). *A semantic analysis of structural recursion.* Diplomarbeit, University of Munich.

Abel, Andreas. (2000). Specification and verification of a formal system for structurally recursive functions. *Types for Proof and Programs, International Workshop, TYPES '99, Selected Papers.* Lecture Notes in Computer Science, vol. 1956. Springer-Verlag.

Abel, Andreas, & Altenkirch, Thorsten. (2000). A predicative strong normalisation proof for a λ-calculus with interleaving inductive types. *Types for Proof and Programs, International Workshop, TYPES '99, Selected Papers.* Lecture Notes in Computer Science, vol. 1956.

Aczel, Peter. (1997). *Notes on constructive set theory.* Published on the WWW.

Amadio, Roberto M., & Coupet-Grimal, Solange. (1998). Analysis of a guard condition in type theory. Nivat, Maurice (ed), *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98.* Lecture Notes in Computer Science, vol. 1378. Springer-Verlag.

Barras, Bruno, *et al.* . (2000). *The Coq proof assistant reference manual.* INRIA. Version 6.3.11.

Benl, Holger. (1998). *Konstruktive Interpretation induktiver Typen.* Diplomarbeit, University of Munich.

Blanqui, Frédéric, Jouannaud, Jean-Pierre, & Okada, Mitsuhiro. (2001). Inductive data type systems. *Theoretical Computer Science*, **277**.

Coquand, Catarina. (1999). *Agda home page.*

Coquand, Thierry. (1992). Pattern matching with dependent types. *Proceedings of the 1992 Workshop on Types for Proofs and Programs.*

Coquand, Thierry, & Huet, Gerard. (1988). The calculus of constructions. *Information and Computation*, **76**, 95 – 120.

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

Dershowitz, Nachum. (1987). Termination of rewriting. *Journal of Symbolic Computation*, **3**, 69–115.

Giesl, Jürgen. (1995). Termination analysis for functional programs using term orderings. *Proceedings of the Second International Static Analysis Symposium.* Lecture Notes in Computer Science, vol. 983. Springer-Verlag.

Gimenéz, Eduardo. (1995). Codifying guarded definitions with recursive schemes. *Pages 39–59 of:* Dybjer, Peter, Nordström, Bengt, & Smith, Jan (eds), *Types for Proofs and Programs, International Workshop TYPES'94, Bøastad, Sweden, June 6-10, 1994, Selected Papers.* Lecture Notes in Computer Science, vol. 996. Springer-Verlag.

Gimenéz, Eduardo. (1998). Structural recursive definitions in type theory. *Pages 397–408 of: Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings.* Lecture Notes in Computer Science, vol. 1443. Springer-Verlag.

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

Gow, Jeremy, Bundy, Alan, & Green, Ian. (1999). Extensions to the estimation calculus. *Pages 258–272 of:* Ganzinger, H., McAllester, D., & Voronkov, A. (eds), *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99).* LNAI, no. 1705. Springer-Verlag.

Harper, Robert. (2000). *Programming in Standard ML.* Carnegie Mellon University.

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

Jones, Mark P, & Reid, Alastair. (1999). *The Hugs 98 user manual.* The Yale Haskell Group and the Oregon Graduate Institute of Science and Technology, http://www.haskell.org/hugs/.

Lee, Chin Soon, Jones, Neil D., & Ben-Amram, Amir M. (2001). The size-change principle for program termination. *ACM Symposium on Principles of Programming Languages.* ACM Press.

Martin-Löf, Per. (1984). *Intuitionistic type theory.* Bibliopolis.

McAllester, David, & Arkoudas, Kostas. (1996). Walther recursion. McRobbie, M. A., & Slaney, J. K. (eds), *13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 – August 3, 1996, Proceedings.* Lecture Notes in Computer Science, vol. 1104. Springer-Verlag.

Nordström, Bengt, Petersson, Kent, & Smith, Jan M. (1990). *Programming in Martin-Löf type theory: An introduction.* Clarendon Press, Oxford.

Pientka, Brigitte, & Pfenning, Frank. (2000). Termination and reduction checking in the Logical Framework. *Workshop on Automation of Proofs by Mathematical Induction, CADE-17, Pittsburgh, PA, USA.*

Tarski, Alfred. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5**, 285 – 309.

Telford, Alistair J., & Turner, David A. (1999). Ensuring termination in ESFP. *15th British Colloquium in Theoretical Computer Science.*

Walther, Christoph. (1988). Argument-bounded algorithms as a basis for automated termination proofs. *Pages 602–621 of:* Lusk, Ewing L., & Overbeek, Ross A. (eds), *9th International Conference on Automated Deduction.* Lecture Notes in Computer Science, vol. 310. Springer-Verlag.