Fixed Points of Type Constructors and Primitive Recursion

Andreas Abel

joint work with Ralph Matthes

CSL'04
Karpacz, Karkonoski National Park, Poland
September 21, 2004

Work supported by: GKLI (DFG), TYPES & APPSEM-II (EU), CoVer (SSF)

Regular Data Types

- Regular data types in Haskell:

```
data Nat    = Zero | Succ Nat
data List a = Nil | Cons a (List a)
```

- Least fixed points of type transformers of kind $* \to *$:

$$\begin{array}{lcl} \mathsf{NatF} & : & * \to * \\ \mathsf{NatF} & := & \lambda X.\, 1 + X \\[1ex] \mathsf{Nat} & : & * \\ \mathsf{Nat} & := & \mu\, \mathsf{NatF} \end{array}$$

- Works also for `List`, since parameter `a` can be abstracted.

$$\begin{array}{lcl} \mathsf{List} & : & * \to * \\ \mathsf{List} & := & \lambda A.\, \mu(\lambda X.\, 1 + A \times X) \end{array}$$

Nested Datatypes

- Non-regular or nested datatype: non-empty triangles.

```
data Tri a = Sg a | Cons a (Tri (e,a))
```

- Parameter (resp., element type) grows in recursion.

$$\begin{array}{c|c|c|c|} A & E & E & E \\ & A & E & E \\ & & A & E \\ & & & A \end{array}$$

- Fixed point of a type *constructor* of kind $(* \to *) \to (* \to *)$ (rank-2 type).

$$\begin{array}{lcl} \mathsf{TriF} & : & (* \to *) \to * \to * \\ \mathsf{TriF} & := & \lambda X \lambda A.\ A + A \times X\,(E \times A) \end{array} \qquad \begin{array}{lcl} \mathsf{Tri} & : & * \to * \\ \mathsf{Tri} & := & \mu\,\mathsf{TriF} \end{array}$$

Programming with Nested Datatypes ...

- ... requires *polymorphic* recursion.
- Example: cutting the top row off a trapezium.

```
cut :: Tri(e,a) -> Tri a

cut (Sg   (e,a)  ) = Sg a
cut (Cons (e,a) r) = Cons a (cut r)
```

- In the recursive call, the argument `r` has type `Tri(e,(e,a))`.
- Does the recursive definition of `cut` have a solution? (Yes.)
- Instance of a *terminating* programming scheme.

Programming Scheme: *Iteration*

- Description:
  - top-down pass: recursive decent into datastructure, adjusting parameters for the ...
  - ... bottom-up pass: composing the result
  - herein: each node treated generically, no access to current position or whole data structure
- Example: `Nat.add`, `List.map`, `List.foldr`
- Properties: termination, computational laws (fusion).
- Drawback: Result is always built from scratch, hence *predecessor* functions like `Nat.pred`, `List.tail` have *linear time* complexity.

Programming Scheme: *Primitive Recursion*

- Primitive recursive functions: e.g., `Nat.factorial` or *redecoration* (Uustalu/Vene, 2002)

```
redec :: (List a -> b) -> List a -> List b
redec f Nil          = Nil
redec f (Cons a as) = Cons (f (Cons a as)) (redec f as)
```

- Like iteration, but access to immediate sublist `as` itself, not just to the result of `redec` for `as`.
- Hence, access to current position `l` = `(Cons a as)` on r.h.s.

The Programming Schemes for Rank 1

- Iteration for rank-1 (= regular) datatypes can be simulated by β-reduction in System F (= $\lambda 2$).
- Primitive recursion can be simulated in an extension Fix (= $\lambda 2U$) of System F by positive fixed point (=retract) types. (Geuvers 1992)

$$\mathsf{It} \longrightarrow \mathsf{F}$$

$$\mathsf{Rec} \longrightarrow \mathsf{Fix}$$

- Primitive recursion *cannot* be simulated by β-reduction in System F. (Spławski/Urzyczyn 1999)

Prim. Rec. for Rank 2: Triangle Redecoration

- Relabelling the diagonal of a triangular matrix: The new diagonal element is computed from its subtriangle by the *redecoration rule* `f :: Tri a -> b`.

```
redec :: (Tri a -> b) -> Tri a -> Tri b
redec f t@(Sg   a  ) = Sg   (f t)
redec f t@(Cons a r) = Cons (f t) (redec (lift f) r)
```

- Herein, we need to lift the redecoration rule to a trapezium.

```
lift :: (Tri a -> b) -> Tri (e,a) -> (e,b)
lift f t = (aux t, f (cut t))
    where aux (Sg   (e,a)  ) = e
          aux (Cons (e,a) r) = e
```

The Programming Schemes for Higher Ranks

- Iteration for rank-n datatypes can be simulated in System F^ω. (TYPES 02, FoSSaCS 03, forthcoming TCS)
- New result: primitive recursion can be simulated in Fix^ω.

$$\mathsf{It}^\omega \longrightarrow \mathsf{F}^\omega$$

$$\mathsf{Rec}^\omega \longrightarrow \mathsf{Fix}^\omega$$

- Fix^ω: System F^ω with fixed points of positive type constructors.
- Difficulty: What is positivity for higher ranks?
- Solution: Distinguish co-/contra-/invariant type constructors by polarity annotation in their kind (Steffen 1998).

System Fix^ω: Syntax

Polarities	p	$::=$	$+$	covariant
		$\mid$	$-$	contravariant
		$\mid$	$\circ$	invariant
Kinds	κ	$::=$	$* \mid p\kappa \to \kappa'$	
Constructors	A, B, F, G	$::=$	$X \mid \lambda X^{p\kappa}.\, F \mid F\,G \mid A \to B \mid \forall X^{\kappa}.\, A \mid \mathsf{fix}\, F$	
Objects (terms)	r, s, t	$::=$	$x \mid \lambda x.t \mid r\; s$	
Contexts	Δ	$::=$	$\diamond \mid \Delta, x : A \mid \Delta, X^{p\kappa}$	

System Fix^ω: Examples

- Impredicative encodings (non-strictly positive):

$$\begin{array}{lcl} \times & : & +* \to +* \to * \\ \times & := & \lambda X^{+*} \lambda Y^{+*} \forall Z^{*}.\, (X \to Y \to Z) \to Z \\ \\ + & : & +* \to +* \to * \\ + & := & \lambda X^{+*} \lambda Y^{+*} \forall Z^{*}.\, (X \to Z) \to (Y \to Z) \to Z \end{array}$$

- Self-composition of monotone $X : +* \to *$ is monotone in X:

$$\lambda X^{+(+*\to*)} \lambda A^{+*}.\, X\,(X\,A) \;:\; +(+* \to *) \to (+* \to *)$$

- But: self-composition of *arbitrary* $X : \circ* \to *$ is *not* monotone in X:

$$\not\vdash \lambda X^{+(\circ*\to*)} \lambda A^{\circ*}.\, X\,(X\,A) \;:\; +(\circ* \to *) \to (\circ* \to *)$$

System Fix^ω: Kinding

- Function space and quantification:

$$\frac{-\Delta \vdash A : * \qquad \Delta \vdash B : *}{\Delta \vdash A \to B : *} \qquad \frac{\Delta, X^{\circ\kappa} \vdash A : *}{\Delta \vdash \forall X^{\kappa}.\, A : *}$$

$-\Delta$ inverts all polarities in Δ.

- Positive fixed points:

$$\frac{\Delta \vdash F : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}\, F : \kappa}$$

- Variables:

$$\frac{X^{p\kappa} \in \Delta \qquad p \in \{+, \circ\}}{\Delta \vdash X : \kappa} \qquad \frac{\Delta, X^{p\kappa} \vdash F : \kappa'}{\Delta \vdash \lambda X^{p\kappa}.\, F : p\kappa \to \kappa'}$$

System Fix$^\omega$: Kinding of Application

- Application of *covariant* constructor:

$$\frac{\Delta \vdash F : +\kappa \to \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

- Application of *contravariant* constructor:

$$\frac{\Delta \vdash F : -\kappa \to \kappa' \qquad -\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

- Application of *invariant* constructor:

$$\frac{\Delta \vdash F : \circ\kappa \to \kappa' \qquad \circ\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

$\circ\Delta$ erases all assumptions with positive or negative polarity from Δ.

System Fix^ω: Equality

- Fixed-point axiom.

$$\frac{\Delta \vdash F : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}\ F = F\,(\mathsf{fix}\ F) : \kappa}$$

- Computation: β-axiom.

$$\frac{\Delta, X^{p\kappa} \vdash F : \kappa' \qquad p\Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^{p\kappa}.\, F)\, G = [G/X]F : \kappa'}$$

- Extensionality: η-axiom.

$$\frac{\Delta \vdash F : p\kappa \to \kappa'}{\Delta \vdash \lambda X^{p\kappa}.\, F\, X = F : \kappa'}\ X \notin \mathsf{FV}(F)$$

- Congruences for all type constructors.
- Symmetry and transitivity. (Reflexivity admissible.)

System Fix^ω: Typing and Reduction

- Typing rules of simply typed lambda-calculus,
- plus quantification,

$$\frac{\Delta, X^{\circ\kappa} \vdash t : A}{\Delta \vdash t : \forall X^\kappa.\, A} \qquad \frac{\Delta \vdash t : \forall X^\kappa.\, A \quad \circ\Delta \vdash F : \kappa}{\Delta \vdash t : [F/X]A}$$

- and type equality (includes fixed point (un)folding).

$$\frac{\Delta \vdash t : A \quad \Delta \vdash A = B : *}{\Delta \vdash t : B}$$

- Reduction: just β.

System Fix^ω: Strong Normalization

- Construct a model of untyped strongly normalizing terms.
- Types are interpreted as saturated set of SN terms, constructors as operators on these sets:

$$
\begin{array}{lcl}
A : * & \Longrightarrow & [\![A]\!] \in \mathsf{SAT} \\
F : +\kappa \to \kappa' & \Longrightarrow & [\![F]\!] \in \mathsf{SAT}^\kappa \xrightarrow{+} \mathsf{SAT}^{\kappa'}
\end{array}
$$

- Positive constructors are interpreted as monotone operators.
- Soundness: If $t : A$ then $t \in [\![A]\!]$.
- Entails that t cannot be reduced infinitely.

Mendler-Style Primitive Recursion

- Natural transformation $F \subseteq^{\vec{\kappa} \to *} G := \forall \vec{X}^{\vec{\kappa}}.\, F\, \vec{X} \to G\, \vec{X}$.
- Formation
$$\mu^{\kappa} : (\kappa \to \kappa) \to \kappa$$
- Introduction
$$\mathsf{in}^{\kappa} : F(\mu^{\kappa} F) \subseteq^{\kappa} \mu^{\kappa} F$$
- Elimination
$$\frac{s : \forall X^{\kappa}.\, (X \subseteq^{\kappa} \mu^{\kappa} F) \to (X \subseteq^{\kappa} G) \to (F\, X \subseteq^{\kappa} G)}{\mathsf{MRec}^{\kappa}\, s : \mu^{\kappa} F \subseteq^{\kappa} G}$$
- Reduction
$$\mathsf{MRec}\, s\, (\mathsf{in}\, t) \longrightarrow_{\beta} s\, \mathsf{id}\, (\mathsf{MRec}\, s)\, t$$
- μ^{κ}, in^{κ}, and MRec^{κ} can be *defined* in Fix^{ω}; the reduction rule is simulated.

On "Conventional" Primitive Recursion

- Conventional primitive recursion relies on monotonicity of type generating functor F.
- For rank 1: $\mathsf{mon} F := \forall A \forall B. (A \to B) \to (F\,A \to F\,B)$.
- For higher ranks: several formulations of monotonicity.
- Basic monotonicity:
  $\mathsf{mon} F := \forall A^\kappa \forall B^\kappa. (A \subseteq^\kappa B) \to (F\,A \subseteq^\kappa F\,B)$.
- But: $\lambda X. X \circ X$ not basic monotone.
- Hence no primitive recursion principle for truly nested datatypes like

```
data Bush a = Nil | Cons a (Bush (Bush a))
```

- Other notions of monotonicity: FoSSaCS 2003, TCS 200?.

Conclusion

Results:

- First formulation (!?) of primitive recursion for nested data types.
- First formulation (!?) of positive recursive types for higher ranks.
- Embedding of primitive recursion into fixed-point types (Geuvers 1992) works also for higher ranks.

Further work: conventional primitive recursion

Related Work

- Nested datatypes: Okasaki 1996, Hinze 1998, Bird/Paterson 1999, Altenkirch/Reus 1999
- Polarized higher-order subtyping: Steffen 1998, Duggan/Compagnoni 1998

Appendix: Semantics of Constructors

Let $\mathsf{U} = \bigcup_\kappa \mathsf{SAT}^\kappa$. For valuation $\theta \in \mathsf{TyVar} \rightharpoonup \mathsf{U}$, define $[\![-]\!]_\theta \in \mathsf{Constr} \rightharpoonup \mathsf{U}$:

$$[\![X]\!]_\theta \quad := \quad \theta(X)$$

$$[\![\lambda X^{p\kappa}.\, F]\!]_\theta \quad := \quad \begin{cases} \mathcal{F} & \text{if } \mathcal{F} \in \mathsf{SAT}^\kappa \xrightarrow{p} \mathsf{SAT}^{\kappa'} \text{ for some } \kappa' \\ \text{undef.} & \text{else} \end{cases}$$

$$\text{where } \mathcal{F}(\mathcal{G} \in \mathsf{SAT}^\kappa) := [\![F]\!]_{\theta[X \mapsto \mathcal{G}]}$$

$$[\![F\, G]\!]_\theta \quad := \quad [\![F]\!]_\theta([\![G]\!]_\theta)$$

$$[\![\mathsf{fix}\, F]\!]_\theta \quad := \quad \begin{cases} \mathsf{lfp}\,\mathcal{F} & \text{if } \mathcal{F} \in \mathsf{SAT}^\kappa \xrightarrow{+} \mathsf{SAT}^{\kappa} \text{ for some } \kappa \\ \text{undef.} & \text{else} \end{cases}$$

$$\text{where } \mathcal{F} := [\![F]\!]_\theta$$

Properties of Semantics

- Extend interpretation to contexts Δ.
- Let $\theta \in \mathsf{SAT}^{\Delta}$ (each variable mapped to semantical operator of correct kind).
- If $\Delta \vdash F : \kappa$ then $[\![F]\!]_\theta \in \mathsf{SAT}^{\kappa}$ (welldefinedness).
- If $\Delta \vdash F = F' : \kappa$ then $[\![F]\!]_\theta = [\![F']\!]_\theta$ (soundness of equality).