Truly nested datatypes through dependent datatypes – a challenge for Coq

Ralph Matthes

Universität München = LMU = University of Munich

Workshop TYPES 2004
Jouy-en-Josas, France
December 15, 2004

Abstract

Following ideas by A. Setzer and P. Aczel, a predicative justification of the truly nested rank-2 datatype $\widehat{\text{Lam}}$ representing untyped lambda-terms that also have a notion of explicit flattening (a weak form of explicit substitution that blocks beta-redexes) is given by an implementation within Coq. The true nesting is turned into a strictly-positive inductive family that is indexed over finite lists of booleans. Implementing the complete beta-developments for the heterogeneous rank-2 datatype Lam for pure lambda-terms by help of $\widehat{\text{Lam}}$ turned out to be a challenge for pattern-matching definitions and the program extraction mechanism of Coq.

Outline

Outline

Outline

heterogeneous/nonregular datatype

Definition (heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Example (powerlists)

In Haskell, define

```
data PList a = Zero a | Succ (PList (a,a))
```

Finite elements of type `Plist a` are ensured by the type system to have a power of 2 elements of type `a`.

Lists are regular: `data List a = Nil | Cons a (List a)`

heterogeneous/nonregular datatype

Definition (heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Example (powerlists)

In Haskell, define

```
data PList a = Zero a | Succ (PList (a,a))
```

Finite elements of type `Plist a` are ensured by the type system to have a power of 2 elements of type `a`.

Lists are regular: `data List a = Nil | Cons a (List a)`

heterogeneous/nonregular datatype

Definition (heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Example (powerlists)

In Haskell, define

```
data PList a = Zero a | Succ (PList (a,a))
```

Finite elements of type `Plist a` are ensured by the type system to have a power of 2 elements of type `a`.

Lists are regular: `data List a = Nil | Cons a (List a)`

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from `a` to `(a,a)`.

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined "stand-alone".
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Outline

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ...  | Constr (Comp (Comp a))
```

yields the typing `Constr :: Comp (Comp a) -> Comp a`

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ...  | Constr (Comp (Comp a))
```

yields the typing `Constr :: Comp (Comp a) -> Comp a`

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ...  | Constr (Comp (Comp a))
```

yields the typing `Constr :: Comp (Comp a) -> Comp a`

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A.

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ...  | Constr (Comp (Comp a))
```

yields the typing `Constr :: Comp (Comp a) -> Comp a`

True nesting is much more difficult to handle.

- Coq does not accept these at all:
  ```
  Non strictly positive occurrence of "Comp" in
  "forall A : Set, Comp (Comp A) -> Comp A"
  ```
- $\lambda F^{Set \to Set}.\, F \circ F : (Set \to Set) \to (Set \to Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases not with structurally smaller arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

True nesting is much more difficult to handle.

- Coq does not accept these at all:
  ```
  Non strictly positive occurrence of "Comp" in
  "forall A : Set, Comp (Comp A) -> Comp A"
  ```
- $\lambda F^{Set \to Set}.\, F \circ F : (Set \to Set) \to (Set \to Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases not with structurally smaller arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

True nesting is much more difficult to handle.

- Coq does not accept these at all:
  ```
  Non strictly positive occurrence of "Comp" in
  "forall A : Set, Comp (Comp A) -> Comp A"
  ```
- $\lambda F^{Set \to Set}. F \circ F : (Set \to Set) \to (Set \to Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases not with structurally smaller arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

True nesting is much more difficult to handle.

- Coq does not accept these at all:
  ```
  Non strictly positive occurrence of "Comp" in
  "forall A : Set, Comp (Comp A) -> Comp A"
  ```
- $\lambda F^{Set \to Set}. F \circ F : (Set \to Set) \to (Set \to Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases not with structurally smaller arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

Outline

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article "Iteration and coiteration schemes for higher-order and nested datatypes"

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article "Iteration and coiteration schemes for higher-order and nested datatypes"

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article "Iteration and coiteration schemes for higher-order and nested datatypes"

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article "Iteration and coiteration schemes for higher-order and nested datatypes"

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article "Iteration and coiteration schemes for higher-order and nested datatypes"

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article "Iteration and coiteration schemes for higher-order and nested datatypes"

Outline

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Outline

Higher-order de Bruijn representation of untyped λ-calculus

We follow Altenkirch/Reus, Bird/Paterson and Fiore/Plotkin/Turi (all 1999).
Lam shall be the least pre-fixed point of the rank 2 "functor"

$$LamF := \lambda X \lambda A.\, A + XA \times XA + X(1 + A).$$

In Coq with impredicative *Set*, this looks like

```
Inductive Lam : Set -> Set :=
    var : forall A : Set, A -> Lam A
  | app : forall A : Set, Lam A -> Lam A -> Lam A
  | abs : forall A : Set, Lam (One + A) -> Lam A
```

Note that the functor *LamF* has to be taken apart in order to get reasonable induction and recursion principles from Coq.

Higher-order de Bruijn representation of untyped λ-calculus

We follow Altenkirch/Reus, Bird/Paterson and Fiore/Plotkin/Turi (all 1999).
Lam shall be the least pre-fixed point of the rank 2 "functor"

$$LamF := \lambda X \lambda A.\, A + XA \times XA + X(1 + A).$$

In Coq with impredicative *Set*, this looks like

```
Inductive Lam : Set -> Set :=
    var : forall A : Set, A -> Lam A
  | app : forall A : Set, Lam A -> Lam A -> Lam A
  | abs : forall A : Set, Lam (One + A) -> Lam A
```

Note that the functor *LamF* has to be taken apart in order to get reasonable induction and recursion principles from Coq.

Higher-order de Bruijn representation of untyped λ-calculus

We follow Altenkirch/Reus, Bird/Paterson and Fiore/Plotkin/Turi (all 1999).
Lam shall be the least pre-fixed point of the rank 2 "functor"

$$LamF := \lambda X \lambda A.\, A + XA \times XA + X(1 + A).$$

In Coq with impredicative *Set*, this looks like

```
Inductive Lam : Set -> Set :=
    var : forall A : Set, A -> Lam A
  | app : forall A : Set, Lam A -> Lam A -> Lam A
  | abs : forall A : Set, Lam (One + A) -> Lam A
```

Note that the functor *LamF* has to be taken apart in order to get reasonable induction and recursion principles from Coq.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B.\quad \underbrace{Lam\, A}_{\text{term argument}} \to \underbrace{(A \to Lam\, B)}_{\text{substitution rule}} \to Lam\, B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B. More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B. \quad \underbrace{Lam\, A}_{\text{term argument}} \quad \to \quad \underbrace{(A \to Lam\, B)}_{\text{substitution rule}} \quad \to Lam\, B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B. More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B.\quad \underbrace{Lam\, A}_{\text{term argument}} \to \underbrace{(A \to Lam\, B)}_{\text{substitution rule}} \to Lam\, B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B. More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B.\quad \underbrace{Lam\, A}_{\text{term argument}} \to \underbrace{(A \to Lam\, B)}_{\text{substitution rule}} \to Lam\, B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B. More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Outline

Formal definition

Formally, just add the rule of *Comp* to *Lam*:

$$\widehat{LamF} := \lambda X \lambda A.\, LamF + X(XA)$$

and call its least pre-fixed point $\widehat{Lam}$.

As data constructors, we need

$$\begin{aligned}
\widehat{var} &: \forall A.\, A \to \widehat{Lam}\, A \\
\widehat{app} &: \forall A.\, \widehat{Lam}\, A \to \widehat{Lam}\, A \to \widehat{Lam}\, A \\
\widehat{abs} &: \forall A.\, \widehat{Lam}\, (1 + A) \to \widehat{Lam}\, A \\
\widehat{flat} &: \forall A.\, \widehat{Lam}\, (\widehat{Lam}\, A) \to \widehat{Lam}\, A
\end{aligned}$$

Formal definition

Formally, just add the rule of *Comp* to *Lam*:

$$\widehat{LamF} := \lambda X \lambda A.\, LamF + X(XA)$$

and call its least pre-fixed point $\widehat{Lam}$.
As data constructors, we need

$$\begin{array}{lcl}
\widehat{var} & : & \forall A.\, A \to \widehat{Lam}\, A \\
\widehat{app} & : & \forall A.\, \widehat{Lam}\, A \to \widehat{Lam}\, A \to \widehat{Lam}\, A \\
\widehat{abs} & : & \forall A.\, \widehat{Lam}\, (1 + A) \to \widehat{Lam}\, A \\
\widehat{flat} & : & \forall A.\, \widehat{Lam}\, (\widehat{Lam}\, A) \to \widehat{Lam}\, A
\end{array}$$

Informal understanding

$\widehat{flat}$ takes a term of type $\widehat{Lam}\,(\widehat{Lam}\,A)$, hence a term whose variables are themselves terms but which are blocked. If such a variable is x, one could write $x\{x := r\}$ for the explicit substitution of r for x in x. Since this calculus only allows this restricted form of explicit substitution, we write $\{r\}$ instead.

Example

$\lambda y.\, y\,\{\lambda z.\, z\,x_1\}\,\{x_2\}$ can be represented by a term of type $\widehat{Lam}(1+1)$ through an intermediary term of type $\widehat{Lam}(1+\widehat{Lam}(1+1))$.

Informal understanding

$\widehat{flat}$ takes a term of type $\widehat{Lam}\,(\widehat{Lam}\,A)$, hence a term whose variables are themselves terms but which are blocked. If such a variable is x, one could write $x\{x := r\}$ for the explicit substitution of r for x in x. Since this calculus only allows this restricted form of explicit substitution, we write $\{r\}$ instead.

Example

$\lambda y.\, y\,\{\lambda z.\, z\,x_1\}\,\{x_2\}$ can be represented by a term of type $\widehat{Lam}(1+1)$ through an intermediary term of type $\widehat{Lam}(1+\widehat{Lam}(1+1))$.

A predicative implementation in Coq

- The construction follows a suggestion by Anton Setzer and a manuscript by Peter Aczel but now adds the algorithmic aspects.
- The idea is to trace how the argument to $\widehat{Lam}$ is transformed during the recursive calls: It can either be augmented by 1, or $\widehat{Lam}$ can be applied (in the case of explicit flattening).
- So we start with a family of inductive types indexed by the initial type A (in a predicative manner since it is just a fixed parameter) and also indexed by a finite list of booleans that encode the type actually given to $\widehat{Lam}$.

A predicative implementation in Coq

- The construction follows a suggestion by Anton Setzer and a manuscript by Peter Aczel but now adds the algorithmic aspects.
- The idea is to trace how the argument to $\widehat{Lam}$ is transformed during the recursive calls: It can either be augmented by 1, or $\widehat{Lam}$ can be applied (in the case of explicit flattening).
- So we start with a family of inductive types indexed by the initial type A (in a predicative manner since it is just a fixed parameter) and also indexed by a finite list of booleans that encode the type actually given to $\widehat{Lam}$.

A predicative implementation in Coq

- The construction follows a suggestion by Anton Setzer and a manuscript by Peter Aczel but now adds the algorithmic aspects.
- The idea is to trace how the argument to $\widehat{Lam}$ is transformed during the recursive calls: It can either be augmented by 1, or $\widehat{Lam}$ can be applied (in the case of explicit flattening).
- So we start with a family of inductive types indexed by the initial type A (in a predicative manner since it is just a fixed parameter) and also indexed by a finite list of booleans that encode the type actually given to $\widehat{Lam}$.

A predicative implementation in Coq, contd.

The actual inductive definition is as follows:

```
Definition index := list bool.
Inductive L (A:Set) : index -> Set :=
  | initL: A -> L A nil
...
  | absL: forall i:index, L A (true::false::i) -> L A (true::i)
  | exsL: forall i:index, L A (true::true::i) -> L A (true::i).
```

Then the slice for index `true::nil` qualifies as $\widehat{Lam}$.
For this, one has to show `L (L A i2) i1 -> L A (i1++i2)` and monotonicity of the whole construction in the argument `A`. By help of these, the data constructors can be defined.

Outline

Recalling developments

One maximal β-development in untyped λ-calculus:

$$\begin{aligned} dev\, x &:= x \\ dev\,(\lambda x.r) &:= \lambda x.\, dev\, r \\ dev\,((\lambda x.r)\, s) &:= (dev\, r)[x := dev\, s] \\ dev\,(r\, s) &:= (dev\, r)(dev\, s) \qquad \text{otherwise} \end{aligned}$$

We do not want to execute the substitution immediately but use an explicit substitution. So we define a function devel of type $\forall A.\, Lam A \to \widehat{Lam}\, A$ and then use a function *eval* of type $\forall A.\, \widehat{Lam}\, A \to Lam\, A$ that resolves the explicit substitutions.

Recalling developments

One maximal β-development in untyped λ-calculus:

$$\begin{aligned} dev\, x &:= x \\ dev\,(\lambda x.r) &:= \lambda x.\, dev\, r \\ dev\,((\lambda x.r)\, s) &:= (dev\, r)[x := dev\, s] \\ dev\,(r\, s) &:= (dev\, r)(dev\, s) \qquad \text{otherwise} \end{aligned}$$

We do not want to execute the substitution immediately but use an explicit substitution. So we define a function devel of type $\forall A.\, Lam A \to \widehat{Lam}\, A$ and then use a function *eval* of type $\forall A.\, \widehat{Lam}\, A \to Lam\, A$ that resolves the explicit substitutions.

A glance at the definition in Coq

The definition in Coq is as follows (thanks to Yves Bertot):

```
Fixpoint devel (A:Set)(t:Lam A){struct t} : Lam' A :=
  match t in Lam A' return Lam' A' with
    | var _ a => var' a
    | abs _ r => abs' (devel r)
    | app  _  r s =>
      let r' := devel r in let s' := devel s in
      match r in Lam A'' return Lam' A'' -> Lam' A''-> Lam' A''
        | abs _ r0 => fun _ y => let r0':= devel r0 in
          esubst r0' (fun z: _ => match z with
                                    | inl _ => y
                                    | inr a => var' a
                                  end)
        | _ => fun x y => (app' x y)
      end r' s'
  end.
```

Experiments with program extraction

- Astonishingly, the construction computes the normal forms for exponentiation of Church numerals.
- Space and time considerations as well as the demand for interoperability with algorithms from other sources dictate the extraction into functional programming languages that are optimized for those purposes.
- The program extraction facility of Coq still needs the use of recursive type equations in Ocaml and sometimes (e. g., for the function devel) it needs an explicit type annotation—despite the fact that already plenty of unsafe coerces take place.

Experiments with program extraction

- Astonishingly, the construction computes the normal forms for exponentiation of Church numerals.
- Space and time considerations as well as the demand for interoperability with algorithms from other sources dictate the extraction into functional programming languages that are optimized for those purposes.
- The program extraction facility of Coq still needs the use of recursive type equations in Ocaml and sometimes (e. g., for the function devel) it needs an explicit type annotation—despite the fact that already plenty of unsafe coerces take place.

Experiments with program extraction

- Astonishingly, the construction computes the normal forms for exponentiation of Church numerals.
- Space and time considerations as well as the demand for interoperability with algorithms from other sources dictate the extraction into functional programming languages that are optimized for those purposes.
- The program extraction facility of Coq still needs the use of recursive type equations in Ocaml and sometimes (e. g., for the function devel) it needs an explicit type annotation—despite the fact that already plenty of unsafe coerces take place.

Outline

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Things still to do

- a general explanation of the method for all nested datatypes
- proof of soundness of the obtained iteration principles

Thank you. Merci.

Typeset with LaTeX using Till Tantau's `beamer.cls`