Radbound Universiteit

A general formulation of simultaneous inductive-recursive definitions in type theory

Peter Dybjer, 1998

Dick Arends and Stephan Stanisic

Radboud University

Contents

- What is simultaneous induction-recursion?
- General schema
- Tarski Universe Construction

What is simultaneous induction-recursion?

Induction-Recursion

Basic Idea: Define a function and its domain at the **same** time.

$$f : D \to R$$

- The function definition is recursive by induction on D,
- and the datatype D depends on f.

Motivating Example

Let's say we are defining a little expression language.

```
Inductive Exp : Set → Set :=
| add        : Exp nat  → Exp nat → Exp nat
| ifthenelse : Exp bool → Exp nat → Exp nat → Exp nat
| lt         : Exp nat  → Exp nat → Exp bool.
```

printf

Now we would like to add `printf` as a function that is callable from our little expression language.

`printf` is a popular function from C for formatting strings:

```
printf("Welcome, %s!\n", "Ulf Norell");
printf("%s, %s!\n",       "Hello", "Catarina Coquand");
printf("%s (%s, %d)\n",  "Data types à la carte", "Swierstra", 2008);
```

```
Welcome, Ulf Norell!
Hello, Catarina Coquand!
Data types à la carte (Swierstra, 2008)
```

Updating our language

So we add printf in our expression type, but what do we put into the hole?

```
Inductive Exp : Set → Set :=
| add        : Exp nat  → Exp nat → Exp nat
| ifthenelse : Exp bool → Exp nat → Exp nat → Exp nat
| lt         : Exp nat  → Exp nat → Exp bool
| printf     : string   → ?       → Exp unit.
```

Generating the type of printf

```
Inductive Exp : Set → Set :=
| add        : Exp nat  → Exp nat → Exp nat
| ifthenelse : Exp bool → Exp nat → Exp nat → Exp nat
| lt         : Exp nat  → Exp nat → Exp bool
| printf     : forall n : string, printftype n → Exp unit
with
Fixpoint printftype (s : string) : Set :=
  ?
```

Generating the type of printf

```
Inductive Exp : Set → Set :=
| add        : Exp nat  → Exp nat → Exp nat
| ifthenelse : Exp bool → Exp nat → Exp nat → Exp nat
| lt         : Exp nat  → Exp nat → Exp bool
| printf     : forall n : string, printftype n → Exp unit
with
Fixpoint printftype (s : string) : Set :=
  match s with
  | "%d" ++ xs ⇒ prod (Exp nat) (printftype xs)
  ?
  end.
```

Generating the type of printf

```
Inductive Exp : Set → Set :=
| add        : Exp nat  → Exp nat → Exp nat
| ifthenelse : Exp bool → Exp nat → Exp nat → Exp nat
| lt         : Exp nat  → Exp nat → Exp bool
| printf     : forall n : string, printftype n → Exp unit
with
Fixpoint printftype (s : string) : Set :=
  match s with
  | "%d" ++ xs ⇒ prod (Exp nat) (printftype xs)
  | String _ xs ⇒ printftype xs
  | EmptyString ⇒ Exp unit
  end.
```

Induction-Recursion

```
Inductive DList (A : Set) : Set :=
| nil : DList A
| cons : forall (b : A) (u : DList), fresh u b → DList A
with
Fixpoint fresh (as : DList A) (a : A) : Set :=
  match as with
  | nil ⇒ true
  | cons b u H ⇒ a ≠ b ∧ fresh u a
  end.
```

General Schema for Induction-Recursion

General Schema

- Formation Rules
- Introduction Rules
- Equality Rules
- Elimination Rules

General Schema : Formation Rules

Formation Rules:

$$P : (A :: \sigma)(a :: \alpha[A])\,\mathrm{set}$$

$$f : \underbrace{(A :: \sigma)}_{\text{parameters}}\underbrace{(a :: \alpha[A])}_{\text{indices}}(c : P(A, a))\psi[A, a]$$

Formation Rules:

$$P : (A :: \sigma)(a :: \alpha[A])\,\mathrm{set}$$

$$f : \underbrace{(A :: \sigma)}_{\text{parameters}}\underbrace{(a :: \alpha[A])}_{\text{indices}}(c : P(A, a))\psi[A, a]$$

$$\mathrm{DList} : (A : \mathrm{set})(\neq : (A)(A)\,\mathrm{set})\,\mathrm{set}$$

$$\mathrm{Fresh} : (A : \mathrm{set})(\neq : (A)(A)\,\mathrm{set})(c : \mathrm{DList})(a : A)\,\mathrm{set}$$

Formation Rules:

$$P : (A :: \sigma)(a :: \alpha[A])\,\mathrm{set}$$

$$f : (A :: \sigma)(a :: \alpha[A])(c : P(A, a))\psi[A, a]$$

$$\underbrace{\mathrm{DList}}_{P} : \underbrace{(A : \mathrm{set})(\neq : (A)(A)\,\mathrm{set})}_{A}\,\mathrm{set}$$

$$\underbrace{\mathrm{Fresh}}_{f} : \underbrace{(A : \mathrm{set})(\neq : (A)(A)\,\mathrm{set})}_{A}\,\underbrace{(c : \mathrm{DList}\ A)}_{c}\,\underbrace{(a : A)\,\mathrm{set}}_{\psi[A,a]}$$

Note: $\alpha[A]$ is the empty sequence.

General Schema : Formation Rules

The previous slide showed explicit parameters, in the rest of the presentation we consider parameters to be implicit.

Resulting in:

$$P : (a :: \alpha)\,\mathrm{set}$$
$$f : (a :: \alpha)(c : P(a))\psi[a]$$

General Schema : Introduction Rules

Introduction Rules:

$$\textit{intro} : \quad \cdots \quad (b : \beta) \quad \cdots \quad (u : (x :: \xi)P(p[x])) \quad \cdots \quad P(q)$$

General Schema : Introduction Rules

Introduction Rules:

$$\textit{intro} : \quad \cdots \quad \underbrace{(b : \beta)}_{\text{non-recursive}} \quad \cdots \quad \underbrace{(u : (x :: \xi) P(p[x]))}_{\text{recursive}} \quad \cdots \quad P(q)$$

General Schema : Introduction Rules

Typing criteria for ξ, p and q are analogous.

$$\textit{intro} : \quad \cdots \quad (b : \beta[\ldots, b', \ldots, u', \ldots]) \quad \cdots (u : (x :: \xi)P(p[x])) \cdots P(q)$$

Here $b' : \beta'$ and $u' : (x' :: \xi')P(p'[x'])$ are non-recursive and recursive earlier premises respectively.

Dependence on earlier recursive premise should only happen through application of f, that is

$$\beta[\ldots, b', \ldots, u', \ldots]$$

must be of the form

$$\hat{\beta}[\ldots, b', \ldots, (x')f(p'[x'], u'(x')), \ldots]$$

General Schema : Introduction Rules

$$\textit{intro}: \quad \cdots \quad (b : \beta) \quad \cdots \quad (u : (x :: \xi)P(p[x])) \quad \cdots \quad P(q)$$

where

🎩 $$\beta[\ldots, b', \ldots, (x')f(p'[x'], u'(x')), \ldots]$$

is a small type in the context

$$(\ldots, b' : \beta', \ldots, v' : (x' :: \xi')\psi[p'[x']], \ldots)$$

General Schema : Introduction Rules

Introduction Rules:

$$\textit{intro} : \cdots (b : \beta) \cdots (u : (x :: \xi) P(p[x])) \cdots (b' : \beta') \cdots P(q)$$

Example:

$$\begin{aligned} \mathrm{nil} &: \mathrm{DList} \\ \mathrm{cons} &: (b : A)(u : \mathrm{DList})(H : \mathrm{Fresh}(u, b))\, \mathrm{DList} \end{aligned}$$

3 premises of which only the second one is recursive.

- $b : A$, non-recursive, $\beta = A$.
- $u : \mathrm{DList}$, recursive, ξ empty and $P = \mathrm{DList}$.
- $H : \mathrm{Fresh}(u, b)$, non-recursive, depends on u (a DList instance, but only through Fresh), $\beta'[b, u] = \hat{\beta}'[b, \mathrm{Fresh}(u)] = \mathrm{Fresh}(u, b)$.

General Schema : Introduction Rules

$$\textit{intro} : \quad \cdots \quad (b : \beta) \quad \cdots \quad (u : (x :: \xi)P(p[x])) \quad \cdots \quad P(q)$$

Note: Removing the dependence of β, ξ, p and q on earlier recursive terms yield the introduction rules we saw in an earlier presentation:

$$\begin{aligned} intro : & \ (A :: \sigma) \\ & \ (b :: \beta[A]) \\ & \ (u :: \gamma[A, b]) \\ & \ P_A(p[A, b]) \end{aligned}$$

Equality Rules:

$$f(q, \mathit{intro}(\ldots, b, \ldots, u, \ldots)) = e(\ldots, b, \ldots, (x)f(p[x], u(x)), \ldots) : \psi[q]$$

Reminder:

$$\mathit{intro} : \cdots \underbrace{(b : \beta)}_{\text{non-recursive}} \cdots \underbrace{(u : (x :: \xi)P(p[x]))}_{\text{recursive}} \cdots P(q)$$

General Schema: Equality Rules

$$f(q, \textit{intro}(\ldots, b, \ldots, u, \ldots)) = e(\ldots, b, \ldots, (x)f(p[x], u(x)), \ldots)$$

in the context

$$(\ldots, b : \beta, \ldots, u : (x :: \xi)P(p[x]), \ldots)$$

where

$$e(\ldots, b, \ldots, v, \ldots) : \psi[q]$$

in the context

$$(\ldots, b : \beta, \ldots, v : (x :: \xi)\psi[p[x]], \ldots)$$

General Schema : Equality Rules

$$f(q, \mathit{intro}(\ldots, b, \ldots, u, \ldots)) = e(\ldots, b, \ldots, (x)f(p[x], u(x)), \ldots)$$

Example:

$$\mathrm{Fresh}(\mathrm{nil}) = (a)\top$$
$$\mathrm{Fresh}(\mathrm{cons}(b, u, H)) = (a)(b \neq a \wedge \mathrm{Fresh}(u, a))$$

General Schema : Elimination Rules

Let P, f be a simultaneously defined inductive type P with recursive function f.

Then we can define a new function g

$$g : (a :: \alpha)(c : P(a))\phi[a, c]$$

using P-recursion.

Elimination:

$$g(q, \textit{intro}(\ldots, b, \ldots, u, \ldots)) = e'(\ldots, b, \ldots, u, (x)g(p[x], u(x)), \ldots)$$

in the context

$$(\ldots, b : \beta, \ldots, u : (x :: \xi)P(p[x]), \ldots)$$

where

$$e'(\ldots, b, \ldots, u, v, \ldots) : \phi[q, \textit{intro}(\ldots, b, \ldots, u, \ldots)]$$

in the context

$$(\ldots, b : \beta, \ldots, u : (x :: \xi)P(p[x]), v : (x :: \xi)\phi[p[x], u(x)], \ldots)$$

General Schema : Elimination Rules

$$g(q, \textit{intro}(\ldots, b, \ldots, u, \ldots)) = e'(\ldots, b, \ldots, u, (x)g(p[x], u(x)), \ldots)$$

Example:

$$\text{length} : (c : \text{DList})\mathbb{N}$$

$$\text{length}(\text{nil}) = 0$$

$$\text{length}(\text{cons}(b, u, H)) = S(\text{length}(u))$$

Tarski Universe Construction

Universes

- Russel style Universe:

  If U denotes a universe, then a term $t : U$ is a type.

- Tarski style Universe:

  Every universe consists of a set of *codes* U and a decoding function T (sometimes also denoted as `el`).

  Universe is a pair (U, T).

Tarski Universe

Example: Universe (U, T) containing types for natural numbers and boolean values:

$$\langle \textit{nat} \rangle : U$$

$$\langle \textit{bool} \rangle : U$$

$$T(\langle \textit{nat} \rangle) = \mathbb{N}$$

$$T(\langle \textit{bool} \rangle) = \mathbb{B}$$

$$3 : \mathbb{N}$$

$$\text{True} : \mathbb{B}$$

Definition of (U_0, T_0)

Goal: Use our induction-recursion framework to construct the first Tarski universe (U_0, T_0).

We need

- Formation rules
- Introduction rules
- Equality rules

(U_0, T_0) Formation rules

$$U_0 : \mathrm{set},$$

$$T_0 : (c : U_0)\,\mathrm{set}$$

(U_0, T_0) Introduction rules

We need a constructor (introduction rule) for every type former in the theory.

Restricting ourselves to Π and equality-types:

$$
\begin{aligned}
\langle \textit{nat} \rangle &: U_0 \\
\langle \textit{bool} \rangle &: U_0 \\
\pi_0 &: (u : U_0)(u' : (x : T_0(u))U_0)U_0 \\
\mathsf{eq}_0 &: (U : U_0)(b, b' : T_0(u))U_0
\end{aligned}
$$

(U_0, T_0) Equality rules

$$
\begin{aligned}
T(\langle \mathit{nat} \rangle) &= \mathbb{N} \\
T(\langle \mathit{bool} \rangle) &= \mathbb{B} \\
T_0(\pi_0(u, u')) &= \Pi(T_0(u), (x)T_0(u'(x))) \\
T_0(\mathsf{eq}_0(u, b, b')) &= \mathrm{Eq}(T_0(u), b, b')
\end{aligned}
$$

Remember:

$$
\begin{aligned}
\langle \mathit{nat} \rangle &: U_0 \\
\langle \mathit{bool} \rangle &: U_0 \\
\pi_0 &: (u : U_0)(u' : (x : T_0(u))U_0)U_0 \\
\mathsf{eq}_0 &: (U : U_0)(b, b' : T_0(u))U_0
\end{aligned}
$$

$$T(\langle \textit{nat} \rangle) = \mathbb{N}$$
$$T(\langle \textit{bool} \rangle) = \mathbb{B}$$
$$T_0(\pi_0(u, u')) = \Pi(T_0(u), (x)T_0(u'(x)))$$
$$T_0(\mathsf{eq}_0(u, b, b')) = \mathrm{Eq}(T_0(u), b, b')$$

$$\Pi x : T_0(u).T_0(u'(x))$$

Remember:

$$\langle \textit{nat} \rangle : U_0$$
$$\langle \textit{bool} \rangle : U_0$$
$$\pi_0 : (u : U_0)(u' : (x : T_0(u))U_0)U_0$$
$$\mathsf{eq}_0 : (U : U_0)(b, b' : T_0(u))U_0$$

Further Universes

Second universe (U_1, T_1).

Analogous to (U_0, T_0), but we now also add U_0 formation.

- Formation Rules:

$$U_1 : \mathrm{set},$$
$$T_1 : (U_1)\,\mathrm{set}$$

- Introduction and Equality Rules:

$$\pi_1 : (u : U_1)(u' : (x : T_1(u))U_1)U_1$$
$$T_1(\pi_1(u, u')) = \Pi(T_1(u), (x)T_1(u'(x)))$$

Further Universes

$$u_{01} : U_1$$

$$T_1(u_{01}) = U_0$$

$$t_{01} : U_0(U_1)$$

$$T_1(t_{01}(b)) = T_0(b)$$

Repeat for $(U_2, T_2), (U_3, T_3), \ldots$

Internalizing Universe Construction

We can internalize the construction of universes using a *simultaneous inductive-recursive* scheme.

$$P = \mathrm{NextU} : (U : \mathrm{set})(T : (U)\,\mathrm{set})\,\mathrm{set},$$
$$f = \mathrm{NextT} : (U : \mathrm{set})(T : (U)\,\mathrm{set})(\mathrm{NextU}(U, T))\,\mathrm{set}$$

Internalizing Universe Construction

We can internalize the construction of universes using a *simultaneous inductive-recursive* scheme.

$$P = \mathrm{NextU} : (U : \mathrm{set})(T : (U)\,\mathrm{set})\,\mathrm{set},$$
$$f = \mathrm{NextT} : (U : \mathrm{set})(T : (U)\,\mathrm{set})(\mathrm{NextU}(U, T))\,\mathrm{set}$$

$$U_{n+1} = \mathrm{NextU}(U_n, T_n)$$
$$T_{n+1} = \mathrm{NextT}(U_n, T_n)$$

Internalizing Universe Construction

We can internalize the construction of universes using a *simultaneous inductive-recursive* scheme.

$$\mathrm{NextU} : \mathrm{set},$$
$$\mathrm{NextT} : (\mathrm{NextU})\,\mathrm{set}$$

Keep in mind, $U : \mathrm{set}$ and $T : (U)\,\mathrm{set}$ exist implicitly.

Internalizing Universe Construction

$$\begin{aligned}
\pi &: (u : U)(u' : (x : T(u))U)U \\
T(\pi(u, u')) &= \Pi(T(u), (x)T(u'(x))) \\
\mathsf{eq} &: (U : U)(b, b' : T(u))U \\
T(\mathsf{eq}(u, b, b')) &= \mathrm{Eq}(T(u), b, b')
\end{aligned}$$

$$\begin{aligned}
* &: \mathrm{NextU} \\
\mathrm{NextT}(*) &= U \\
t &: (b : U)\,\mathrm{NextU} \\
\mathrm{NextT}(t(b)) &= T(b)
\end{aligned}$$

Super Universes

Super universe U_∞ is the closure of the next universe operator **and** all other type formers.

Formation Rules:

$$U_\infty : \mathrm{set}$$

$$T_\infty : (U_\infty)\,\mathrm{set}$$

Note: Construction looks very much like the first universe construction.

Super Universe

$$u_0 : U_\infty,$$

$$T_\infty(u_0) = U_0,$$

$$\mathrm{NextU} : (u : U_\infty)(u' : (T_\infty(u))U_\infty)U_\infty,$$

$$T_\infty(\mathrm{NextU}(u, u')) = \mathrm{NextU}(T_\infty(u), (x)T_\infty(u'(x)))$$

Conclusion

Simultaneous induction-recursion is a powerful concept allowing to create more expressive constructions.

We showed:

- The basic idea behind simultaneous induction-recursion.
- A schema to construct simultaneous inductive-recursive definitions.
- How to construct universes (and universe hierarchies) using induction-recursion.

Questions?