Existential Types

Ralph Debusmann

rade@ps.uni-sb.de

Programming Systems Lab

Universität des Saarlandes

Overview

1. Typing and evaluation rules
2. Encoding existential types
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) Introducing objects
   (c) Objects vs. ADTs

Overview

1. Typing and evaluation rules
2. Encoding existential types
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) Introducing objects
   (c) Objects vs. ADTs

Existential introduction

- an existentially typed value introduced by pairing a type with a term: $\langle \mathtt{S}, \mathtt{t} \rangle$

Existential introduction

- an existentially typed value introduced by pairing a type with a term: $\langle S, \mathtt{t} \rangle$
- intuition: value $\langle S, \mathtt{t} \rangle$ of type $\exists X.T$ is a *module* with a *type component* S and a *term component* $\mathtt{t}$, where $[S/X]T$.

Example

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\mathtt{succ}(\mathtt{x})\}\rangle : \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{X}\}$$

Example

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\mathtt{succ}(\mathtt{x})\}\rangle : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{X}\}$$

type component

Example

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\mathtt{succ}(\mathtt{x})\}\rangle : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{X}\}$$

type component

term component

Typing rule for existential introduction

$$\frac{}{\Gamma \vdash \langle \mathsf{S}, \mathtt{t} \rangle : \exists \mathsf{X}.\mathsf{T}} \quad \text{T-PACK'}$$

Typing rule for existential introduction

$$\frac{\Gamma \vdash \mathtt{t} : [\mathsf{S}/\mathsf{X}]\mathsf{T}}{\Gamma \vdash \langle \mathsf{S}, \mathtt{t} \rangle : \exists \mathsf{X}.\mathsf{T}} \quad \text{T-PACK'}$$

Typing rule for existential introduction

$$\frac{\Gamma \vdash \mathtt{t} : [\mathsf{S}/\mathsf{X}]\mathsf{T}}{\Gamma \vdash \langle \mathsf{S}, \mathtt{t} \rangle : \exists \mathsf{X}.\mathsf{T}} \quad \text{T-PACK'}$$

- we lose the unique type property, e.g.:

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\mathtt{succ}(\mathtt{x})\} \rangle : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{X}\}$$

Typing rule for existential introduction

$$\frac{\Gamma \vdash \mathtt{t} : [\mathsf{S}/\mathsf{X}]\mathsf{T}}{\Gamma \vdash \langle \mathsf{S}, \mathtt{t} \rangle : \exists \mathsf{X}.\mathsf{T}} \quad \text{T-PACK'}$$

- we lose the unique type property, e.g.:

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\mathtt{succ}(\mathtt{x})\}\rangle : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{X}\}$$

- can also have type:

$$\exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

Typing rule for existential introduction

$$\frac{\Gamma \vdash t : [S/X]T}{\Gamma \vdash \langle S, t\rangle : \exists X.T} \quad \text{T-PACK'}$$

- we lose the unique type property, e.g.:

$$\langle \mathsf{Int}, \{a = 5, f = \lambda x : \mathsf{Int}.succ(x)\}\rangle : \exists X.\{a : X, f : X \to X\}$$

- can also have type:

$$\exists X.\{a : X, f : X \to \mathsf{Int}\}$$

Typing rule for existential introduction

$$\frac{\Gamma \vdash t : [S/X]T}{\Gamma \vdash \langle S, t\rangle : \exists X.T} \quad \text{T-PACK'}$$

- we lose the unique type property, e.g.:

$$\langle \mathsf{Int}, \{a = 5, f = \lambda x : \mathsf{Int}.succ(x)\}\rangle : \exists X.\{a : X, f : X \rightarrow X\}$$

- can also have type:

$$\exists X.\{a : X, f : X \rightarrow \mathsf{Int}\}$$

- solution: make type annotation mandatory

Type annotation

- e.g.:

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{X}\}$$

Type annotation

- e.g.:

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathsf{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{X}\}$$

$$\langle \mathsf{Int}, \{\mathtt{a} = 5, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathsf{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

Revised typing rule for existential introduction

$$\frac{\Gamma \vdash \mathtt{t} : [\mathsf{S}/\mathsf{X}]\mathsf{T}}{\Gamma \vdash \langle \mathsf{S}, \mathtt{t} \rangle \text{ as } \exists \mathsf{X}.\mathsf{T} : \exists \mathsf{X}.\mathsf{T}} \quad \text{T-PACK}$$

Revised typing rule for existential introduction

$$\frac{\Gamma \vdash \mathtt{t} : [\mathsf{S}/\mathsf{X}]\mathsf{T}}{\Gamma \vdash \langle \mathsf{S}, \mathtt{t} \rangle \ \mathtt{as}\ \exists \mathsf{X}.\mathsf{T} : \exists \mathsf{X}.\mathsf{T}} \quad \text{T-PACK}$$

Existential elimination

- an existentially typed value $\mathtt{m}$ is eliminated by binding its type and term components to variables X and $\mathtt{x}$, and use them in calculating $\mathtt{t}_2$:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{m}\ \mathtt{in}\ \mathtt{t}_2$$

Example

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \longrightarrow \mathsf{Int}\}$$

Example

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ (\lambda \mathtt{y} : \mathsf{X}.\ (\mathtt{x.f}\ \mathtt{y}))\ \mathtt{x.a};$$

Example

$$\texttt{m4} = \langle \mathsf{Int}, \{\texttt{a} = 0, \texttt{f} = \lambda\texttt{x} : \mathsf{Int}.\ \texttt{succ}(\texttt{x})\}\rangle \ \texttt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

$$\texttt{open}\ \langle \mathsf{X}, \texttt{x}\rangle = \texttt{m4}\ \texttt{in}\ (\lambda\texttt{y} : \mathsf{X}.\ (\texttt{x.f}\ \texttt{y}))\ \texttt{x.a};$$

$$\triangleright\ 1 : \mathsf{Int}$$

Typing rule for existential elimination

$$\frac{}{\Gamma \vdash \texttt{open}\ \langle \mathsf{X}, \texttt{x} \rangle = \texttt{t}_1\ \texttt{in}\ \texttt{t}_2 : \mathsf{T}_2} \quad \text{T-UNPACK'}$$

Typing rule for existential elimination

$$\frac{\Gamma \vdash \mathtt{t}_1 : \exists \mathsf{X}.\mathsf{T}_1}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}_2} \quad \text{T-UNPACK'}$$

Typing rule for existential elimination

$$\frac{\Gamma \vdash \mathtt{t}_1 : \exists \mathsf{X}.\mathsf{T}_1 \quad \Gamma, \mathsf{X}, \mathtt{x} : \mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}_2} \quad \text{T-UNPACK'}$$

Evaluation rule

$$\texttt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = (\langle \mathsf{S}, \mathtt{t} \rangle\ \texttt{as}\ \mathsf{T}_1)\ \texttt{in}\ \mathtt{t}_2$$

Evaluation rule

$$\texttt{open}\ \langle \mathsf{X}, \texttt{x} \rangle = (\langle \mathsf{S}, \texttt{t} \rangle\ \texttt{as}\ \mathsf{T}_1)\ \texttt{in}\ \texttt{t}_2 \quad \longrightarrow$$

Evaluation rule

$$\texttt{open}\ \langle \mathsf{X}, \texttt{x} \rangle = (\langle \mathsf{S}, \texttt{t} \rangle\ \texttt{as}\ \mathsf{T}_1)\ \texttt{in}\ \texttt{t}_2 \quad \longrightarrow \quad [\mathsf{S}/\mathsf{X}][\texttt{t}/\texttt{x}]\texttt{t}_2$$

Example

- all operations on term `x` must be warranted by its abstract type, e.g. we cannot use `x.a` concretely as a number (since the concrete type of the module is hidden):

Example

- all operations on term $\mathtt{x}$ must be warranted by its abstract type, e.g. we cannot use $\mathtt{x.a}$ concretely as a number (since the concrete type of the module is hidden):

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

Example

- all operations on term $\mathtt{x}$ must be warranted by its abstract type, e.g. we cannot use $\mathtt{x.a}$ concretely as a number (since the concrete type of the module is hidden):

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{succ}(\mathtt{x.a});$$

Example

- all operations on term $\mathtt{x}$ must be warranted by its abstract type, e.g. we cannot use $\mathtt{x.a}$ concretely as a number (since the concrete type of the module is hidden):

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = \mathtt{0}, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{succ}(\mathtt{x.a});$$

$$\triangleright\ \mathtt{Error:\ argument\ of\ succ\ is\ not\ a\ number}$$

Scoping errors

- be careful:

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda\mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a};$$

Scoping errors

- be careful:

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathsf{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x.a};$$
$$\triangleright\ \mathtt{Error}:\ \mathtt{scoping\ error}$$

Scoping errors

- be careful:

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x.a};$$
$$\triangleright\ \mathtt{Error:\ scoping\ error}$$

- why? consider:

$$\frac{\Gamma \vdash \mathtt{m4} : \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\} \qquad \Gamma, \mathsf{X}, \mathtt{x} \vdash \mathtt{x.a} : \mathsf{X}}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x.a} : \mathsf{X}}$$

Scoping errors

- be careful:

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathsf{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a};$$
$$\triangleright\ \mathtt{Error:}\ \mathtt{scoping\ error}$$

- why? consider:

$$\frac{\Gamma \vdash \mathtt{m4} : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\} \qquad \Gamma, \mathsf{X}, \mathtt{x} \vdash \mathtt{x}.\mathsf{a} : \mathsf{X}}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a} : \mathsf{X}}$$

Scoping errors

- be careful:

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a};$$

$$\triangleright\ \mathtt{Error:}\ \ \mathtt{scoping\ error}$$

- why? consider:

$$\frac{\Gamma \vdash \mathtt{m4} : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\} \qquad \Gamma, \mathsf{X}, \mathtt{x} \vdash \mathtt{x}.\mathsf{a} : \mathsf{X}}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a} : \mathsf{X}}$$

Scoping errors

- be careful:

$$\mathtt{m4} = \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathsf{f} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\rangle\ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a};$$

$$\triangleright\ \mathtt{Error:\ scoping\ error}$$

- why? consider:

$$\frac{\Gamma \vdash \mathtt{m4} : \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\} \quad \Gamma, \mathsf{X}, \mathtt{x} \vdash \mathtt{x}.\mathsf{a} : \mathsf{X}}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m4}\ \mathtt{in}\ \mathtt{x}.\mathsf{a} : \mathsf{X}}$$

- must add side condition to typing rule for existential elimination; X may not occur in the result type

Revised typing rule for existential elimination

$$\frac{\Gamma \vdash \mathtt{t}_1 : \exists \mathsf{X}.\mathsf{T}_1 \quad \Gamma, \mathsf{X}, \mathtt{x} : \mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}_2} \quad \text{T-UNPACK}$$

Revised typing rule for existential elimination

$$\frac{\Gamma \vdash \mathtt{t}_1 : \exists \mathsf{X}.\mathsf{T}_1 \quad \Gamma, \mathsf{X}, \mathtt{x} : \mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{T}_2 \quad \mathsf{X} \notin FV(\mathsf{T}_2)}{\Gamma \vdash \mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}_2} \quad \text{T-UNPACK}$$

Overview

1. Typing and evaluation rules
2. **Encoding existential types**
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) Introducing objects
   (c) Objects vs. ADTs

Duality

- universal types: $\forall X.T$ is a value of type $[S/X]T$ for *all* types S.

Duality

- universal types: $\forall X.T$ is a value of type $[S/X]T$ for *all* types S.
- existential types: $\exists X.T$ is a value of type $[S/X]T$ for *some* type S.

Duality

- universal types: $\forall X.T$ is a value of type $[S/X]T$ for *all* types S.
- existential types: $\exists X.T$ is a value of type $[S/X]T$ for *some* type S.
- idea: exploit duality to encode existential types using universal types, using the equality:

$$\exists X.T \quad = \quad \neg \forall X. \neg T$$

Encoding

- encoding existential types using universal types:

$$\exists X.T \quad \stackrel{\text{def}}{=} \quad \forall Y.(\forall X.T \rightarrow Y) \rightarrow Y$$

Encoding

- encoding existential types using universal types:

$$\exists \mathsf{X}.\mathsf{T} \quad \stackrel{\text{def}}{=} \quad \forall \mathsf{Y}.(\forall \mathsf{X}.\mathsf{T} \to \mathsf{Y}) \to \mathsf{Y}$$

- operational view: a module is a value that gets a result type and a continuation, then calls the continuation to yield the £nal result

Encoding

- encoding existential types using universal types:

$$\exists X.T \quad \stackrel{\text{def}}{=} \quad \forall Y.(\forall X.T \rightarrow Y) \rightarrow Y$$

- operational view: a module is a value that gets a result type and a continuation, then calls the continuation to yield the £nal result

Encoding

- encoding existential types using universal types:

$$\exists \mathsf{X}.\mathsf{T} \quad \stackrel{\text{def}}{=} \quad \forall \mathsf{Y}.(\forall \mathsf{X}.\mathsf{T} \to \mathsf{Y}) \to \mathsf{Y}$$

- operational view: a module is a value that gets a result type and a continuation, then calls the continuation to yield the £nal result

Encoding existential elimination

- given:

$$\texttt{open}\ \langle X, \texttt{x} \rangle = \texttt{t}_1\ \texttt{in}\ \texttt{t}_2$$

  where $\texttt{t}_1 : \forall Y.(\forall X.T \to Y) \to Y$

Encoding existential elimination

- given:

$$\texttt{open}\ \langle X, \texttt{x}\rangle = \texttt{t}_1\ \texttt{in}\ \texttt{t}_2$$

  where $\texttt{t}_1 : \forall Y.(\forall X.T \to Y) \to Y$

- £rst apply to result type T_2 to get type $(\forall X.T \to T_2) \to T_2$:

$$\texttt{open}\ \langle X, \texttt{x}\rangle = \texttt{t}_1\ \texttt{in}\ \texttt{t}_2 \quad \stackrel{\text{def}}{=} \quad \texttt{t}_1\ T_2 \ldots$$

Encoding existential elimination

- given:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$$

  where $\mathtt{t}_1 : \forall \mathsf{Y}.(\forall \mathsf{X}.\mathsf{T} \to \mathsf{Y}) \to \mathsf{Y}$
- £rst apply to result type T_2 to get type $(\forall \mathsf{X}.\mathsf{T} \to \mathsf{T}_2) \to \mathsf{T}_2$:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \quad \stackrel{\text{def}}{=} \quad \mathtt{t}_1\ \mathsf{T}_2 \ldots$$

- then apply to continuation of type $\forall \mathsf{X}.\mathsf{T} \to \mathsf{T}_2$ to get result type T_2:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \quad \stackrel{\text{def}}{=} \quad \mathtt{t}_1\ \mathsf{T}_2\ (\lambda \mathsf{X}.\lambda \mathtt{x} : \mathsf{T}.\mathtt{t}_2)$$

Encoding existential introduction

- given:

$$\langle S, \mathtt{t} \rangle \ \mathtt{as} \ \exists X.T$$

we must use S and $\mathtt{t}$ to build a value of type $\forall Y.(\forall X.T \to Y) \to Y$

Encoding existential introduction

- given:

$$\langle \mathsf{S}, \mathtt{t} \rangle \; \mathtt{as} \; \exists \mathsf{X}.\mathsf{T}$$

  we must use S and $\mathtt{t}$ to build a value of type $\forall \mathsf{Y}.(\forall \mathsf{X}.\mathsf{T} \longrightarrow \mathsf{Y}) \longrightarrow \mathsf{Y}$

- begin with two abstractions:

$$\langle \mathsf{S}, \mathtt{t} \rangle \; \mathtt{as} \; \exists \mathsf{X}.\mathsf{T} \quad \stackrel{\text{def}}{=} \quad \lambda \mathsf{Y}.\lambda \mathtt{f} : (\forall \mathsf{X}.\mathsf{T} \longrightarrow \mathsf{Y}) \ldots$$

Encoding existential introduction

- given:

$$\langle \mathsf{S}, \mathtt{t} \rangle \ \mathtt{as} \ \exists \mathsf{X}.\mathsf{T}$$

we must use S and $\mathtt{t}$ to build a value of type $\forall \mathsf{Y}.(\forall \mathsf{X}.\mathsf{T} \longrightarrow \mathsf{Y}) \longrightarrow \mathsf{Y}$

- begin with two abstractions:

$$\langle \mathsf{S}, \mathtt{t} \rangle \ \mathtt{as} \ \exists \mathsf{X}.\mathsf{T} \quad \stackrel{\text{def}}{=} \quad \lambda \mathsf{Y}.\lambda \mathtt{f} : (\forall \mathsf{X}.\mathsf{T} \longrightarrow \mathsf{Y}) \ldots$$

- apply $\mathtt{f}$ to appropriate arguments: £rst, supply S:

$$\langle \mathsf{S}, \mathtt{t} \rangle \ \mathtt{as} \ \exists \mathsf{X}.\mathsf{T} \quad \stackrel{\text{def}}{=} \quad \lambda \mathsf{Y}.\lambda \mathtt{f} : (\forall \mathsf{X}.\mathsf{T} \longrightarrow \mathsf{Y}).\ \mathtt{f}\ \mathsf{S} \ldots$$

Encoding existential introduction

- given:

$$\langle S, \mathtt{t} \rangle \ \mathtt{as} \ \exists X.T$$

we must use S and $\mathtt{t}$ to build a value of type $\forall Y.(\forall X.T \to Y) \to Y$

- begin with two abstractions:

$$\langle S, \mathtt{t} \rangle \ \mathtt{as} \ \exists X.T \quad \stackrel{\text{def}}{=} \quad \lambda Y.\lambda \mathtt{f} : (\forall X.T \to Y) \ldots$$

- apply $\mathtt{f}$ to appropriate arguments: £rst, supply S:

$$\langle S, \mathtt{t} \rangle \ \mathtt{as} \ \exists X.T \quad \stackrel{\text{def}}{=} \quad \lambda Y.\lambda \mathtt{f} : (\forall X.T \to Y).\ \mathtt{f}\ S \ldots$$

- then supply $\mathtt{t}$ of type S to get result type Y:

$$\langle S, \mathtt{t} \rangle \ \mathtt{as} \ \exists X.T \quad \stackrel{\text{def}}{=} \quad \lambda Y.\lambda \mathtt{f} : (\forall X.T \to Y).\ \mathtt{f}\ S\ \mathtt{t}$$

Overview

1. Typing and evaluation rules
2. Encoding existential types
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) Introducing objects
   (c) Objects vs. ADTs

Parametricity

- consider:

$$\mathtt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \longrightarrow \mathsf{Int}\}$$

Parametricity

- consider:

$$\texttt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathsf{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\}\rangle \ \texttt{as} \ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\texttt{m2} \quad = \quad \langle \mathsf{Bool}, \{\mathsf{a} = \texttt{false}, \mathsf{f} = \lambda \mathtt{x} : \mathsf{Bool}.\ 0\}\rangle \ \texttt{as} \ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

Parametricity

- consider:

$$\mathtt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{m2} \quad = \quad \langle \mathsf{Bool}, \{\mathtt{a} = \mathtt{false}, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Bool}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

- evaluation does not depend on the speci£c type of `m1` and `m2`: it is parametric in X:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m1}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$

Parametricity

- consider:

$$\texttt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\}\rangle \ \texttt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

$$\texttt{m2} \quad = \quad \langle \mathsf{Bool}, \{\mathsf{a} = \texttt{false}, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Bool}.\ 0\}\rangle \ \texttt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

- evaluation does not depend on the speci£c type of `m1` and `m2`: it is parametric in X:

$$\texttt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \texttt{m1}\ \texttt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$
$$\triangleright\ 0$$

Parametricity

- consider:

$$\mathtt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathtt{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{m2} \quad = \quad \langle \mathsf{Bool}, \{\mathtt{a} = \mathtt{false}, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Bool}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathtt{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \to \mathsf{Int}\}$$

- evaluation does not depend on the speci£c type of m1 and m2: it is parametric in X:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m1}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$
$$\triangleright\ 0$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m2}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$

Parametricity

- consider:

$$\mathtt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

$$\mathtt{m2} \quad = \quad \langle \mathsf{Bool}, \{\mathsf{a} = \mathtt{false}, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Bool}.\ 0\}\rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathsf{f} : \mathsf{X} \to \mathsf{Int}\}$$

- evaluation does not depend on the speci£c type of `m1` and `m2`: it is parametric in X:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m1}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$
$$\triangleright\ 0$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x}\rangle = \mathtt{m2}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$
$$\triangleright\ 0$$

Parametricity

- consider:

$$\mathtt{m1} \quad = \quad \langle \mathsf{Int}, \{\mathsf{a} = 0, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Int}.\ 0\} \rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

$$\mathtt{m2} \quad = \quad \langle \mathsf{Bool}, \{\mathsf{a} = \mathtt{false}, \mathtt{f} = \lambda \mathtt{x} : \mathsf{Bool}.\ 0\} \rangle \ \mathtt{as}\ \exists \mathsf{X}.\{\mathsf{a} : \mathsf{X}, \mathtt{f} : \mathsf{X} \rightarrow \mathsf{Int}\}$$

- evaluation does not depend on the speci£c type of `m1` and `m2`: it is parametric in X:

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{m1}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$
$$\triangleright\ 0$$

$$\mathtt{open}\ \langle \mathsf{X}, \mathtt{x} \rangle = \mathtt{m2}\ \mathtt{in}\ (\mathtt{x.f}\ \mathtt{x.a})$$
$$\triangleright\ 0$$

- idea: use parametricity to construct two kinds of programmer de£ned abstractions: abstract data types (ADTs) and objects

Overview

1. Typing and evaluation rules
2. Encoding existential types
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) Introducing objects
   (c) Objects vs. ADTs

SML example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

SML example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

abstract representation type

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

SML example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

abstract representation type (→ `type counter` in the signature)

concrete representation type (→ `int` in `type counter = int` in the structure)

SML example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

abstract representation type

concrete representation type

interface

SML example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

abstract representation type

concrete representation type

interface

implementation

Example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

```
- counter.get (counter.inc counter.new);
```

Example

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

```
- counter.get (counter.inc counter.new);
val it = 2 : int
```

ADTs as existentials

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

ADTs as existentials

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

$$\begin{aligned}&\mathsf{COUNTER} = \\ &\exists \mathsf{Counter}.\{\\ &\quad \mathsf{new} : \mathsf{Counter},\\ &\quad \mathsf{get} : \mathsf{Counter} \rightarrow \mathsf{Int},\\ &\quad \mathsf{inc} : \mathsf{Counter} \rightarrow \mathsf{Counter}\}\end{aligned}$$

ADTs as existentials

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

$$\begin{aligned}
&\mathsf{COUNTER} = \\
&\exists \mathsf{Counter}.\{ \\
&\quad \mathsf{new} : \mathsf{Counter}, \\
&\quad \mathsf{get} : \mathsf{Counter} \to \mathsf{Int}, \\
&\quad \mathsf{inc} : \mathsf{Counter} \to \mathsf{Counter}\}
\end{aligned}$$

$$\begin{aligned}
&\mathtt{CounterADT} = \\
&\quad \langle \mathsf{Int}, \{ \\
&\quad\quad \mathtt{new} = 1, \\
&\quad\quad \mathtt{get} = \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{n} \\
&\quad\quad \mathtt{inc} = \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{succ}(\mathtt{n})\}\rangle \\
&\mathsf{as}\ \mathsf{COUNTER}
\end{aligned}$$

ADTs as existentials

```
signature COUNTER =
sig
  type counter
  val new : counter
  val get : counter → int
  val inc : counter → counter
end;
```

```
structure Counter :> COUNTER =
struct
  type counter = int
  val new = 1
  fun get(n) = n
  fun inc(n) = n + 1
end;
```

$$\begin{array}{l} \mathsf{COUNTER} = \\ \exists \mathsf{Counter}.\{ \\ \quad \mathsf{new} : \mathsf{Counter}, \\ \quad \mathsf{get} : \mathsf{Counter} \to \mathsf{Int}, \\ \quad \mathsf{inc} : \mathsf{Counter} \to \mathsf{Counter}\} \end{array}$$

$$\begin{array}{l} \mathtt{CounterADT} = \\ \quad \langle \mathsf{Int}, \{ \\ \quad \mathtt{new} = 1, \\ \quad \mathtt{get} = \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{n} \\ \quad \mathtt{inc} = \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{succ}(\mathtt{n})\}\rangle \\ \mathsf{as\ COUNTER} \end{array}$$

- Mitchell/Plotkin 1984: “Abstract types have existential type”

Example

$$\begin{aligned}
&\mathsf{COUNTER} = \\
&\exists \mathsf{Counter}.\{ \\
&\quad \mathsf{new} \ : \ \mathsf{Counter}, \\
&\quad \mathsf{get} \ : \ \mathsf{Counter} \to \mathsf{Int}, \\
&\quad \mathsf{inc} \ : \ \mathsf{Counter} \to \mathsf{Counter}\}
\end{aligned}$$

$$\begin{aligned}
&\mathtt{CounterADT} = \\
&\quad \langle \mathsf{Int}, \{ \\
&\quad \mathtt{new} \ = \ \mathtt{1}, \\
&\quad \mathtt{get} \ = \ \lambda \mathtt{n} : \mathsf{Int}.\ \mathtt{n} \\
&\quad \mathtt{inc} \ = \ \lambda \mathtt{n} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{n})\}\rangle \\
&\mathtt{as}\ \mathsf{COUNTER}
\end{aligned}$$

Example

$$\begin{aligned}&\mathsf{COUNTER} =\\ &\exists \mathsf{Counter}.\{\\ &\quad \mathsf{new} \;:\; \mathsf{Counter},\\ &\quad \mathsf{get} \;:\; \mathsf{Counter} \to \mathsf{Int},\\ &\quad \mathsf{inc} \;:\; \mathsf{Counter} \to \mathsf{Counter}\}\end{aligned}$$

$$\begin{aligned}&\mathtt{CounterADT} =\\ &\quad \langle \mathsf{Int}, \{\\ &\quad\; \mathtt{new} \;=\; 1,\\ &\quad\; \mathtt{get} \;=\; \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{n}\\ &\quad\; \mathtt{inc} \;=\; \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{succ}(\mathtt{n})\}\rangle\\ &\mathtt{as}\ \mathsf{COUNTER}\end{aligned}$$

$$\begin{aligned}&\mathtt{open}\ \langle \mathsf{Counter}, \mathtt{counter}\rangle = \mathtt{CounterADT}\ \mathtt{in}\\ &\mathtt{counter.get}\ (\mathtt{counter.inc}\ \mathtt{counter.new});\end{aligned}$$

Example

$$
\begin{aligned}
&\mathsf{COUNTER} = \\
&\exists \mathsf{Counter}.\{ \\
&\quad \mathsf{new} \;:\; \mathsf{Counter}, \\
&\quad \mathsf{get} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Int}, \\
&\quad \mathsf{inc} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Counter}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{CounterADT} = \\
&\quad \langle \mathsf{Int}, \{ \\
&\quad\quad \mathtt{new} \;=\; 1, \\
&\quad\quad \mathtt{get} \;=\; \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{n} \\
&\quad\quad \mathtt{inc} \;=\; \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{succ}(\mathtt{n})\}\rangle \\
&\mathtt{as}\ \mathsf{COUNTER}
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{open}\ \langle \mathsf{Counter}, \mathtt{counter}\rangle = \mathtt{CounterADT}\ \mathtt{in} \\
&\mathtt{counter.get}\ (\mathtt{counter.inc}\ \mathtt{counter.new}); \\
&\triangleright\ 2 : \mathsf{Int}
\end{aligned}
$$

Example

$$\mathsf{COUNTER} = \exists \mathsf{Counter}.\{\ \mathsf{new} : \mathsf{Counter},\ \mathsf{get} : \mathsf{Counter} \rightarrow \mathsf{Int},\ \mathsf{inc} : \mathsf{Counter} \rightarrow \mathsf{Counter}\}$$

$$\mathtt{CounterADT} = \langle \mathsf{Int}, \{\ \mathtt{new} = 1,\ \mathtt{get} = \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{n},\ \mathtt{inc} = \lambda \mathtt{n} : \mathsf{Int}.\, \mathtt{succ}(\mathtt{n})\}\rangle \ \mathtt{as}\ \mathsf{COUNTER}$$

$$\mathtt{open}\ \langle \mathsf{Counter}, \mathtt{counter}\rangle = \mathtt{CounterADT}\ \mathtt{in}$$
$$\mathtt{counter.get}\ (\mathtt{counter.inc}\ \mathtt{counter.new});$$
$$\triangleright\ 2 : \mathsf{Int}$$

- type name $\mathsf{Counter}$ can be used just like a new base type

Example

$$\mathsf{COUNTER} = \exists\mathsf{Counter}.\{\ \mathsf{new} : \mathsf{Counter},\ \mathsf{get} : \mathsf{Counter} \to \mathsf{Int},\ \mathsf{inc} : \mathsf{Counter} \to \mathsf{Counter}\}$$

$$\mathtt{CounterADT} = \langle \mathsf{Int}, \{\ \mathtt{new} = 1,\ \mathtt{get} = \lambda\mathtt{n} : \mathsf{Int}.\,\mathtt{n},\ \mathtt{inc} = \lambda\mathtt{n} : \mathsf{Int}.\,\mathtt{succ}(\mathtt{n})\}\rangle\ \mathtt{as}\ \mathsf{COUNTER}$$

$$\mathtt{open}\ \langle\mathsf{Counter}, \mathtt{counter}\rangle = \mathtt{CounterADT}\ \mathtt{in}$$
$$\mathtt{counter.get}\ (\mathtt{counter.inc}\ \mathtt{counter.new});$$
$$\triangleright\ 2 : \mathsf{Int}$$

- type name Counter can be used just like a new base type
- e.g. we can de£ne new ADTs with representation type Counter, e.g. a ¤ip-¤op

Flip-flop

$$
\begin{aligned}
&\texttt{open}\ \langle \mathsf{Counter}, \texttt{counter}\rangle = \texttt{CounterADT in} \\
&\texttt{FlipFlopADT} = \\
&\quad \langle \mathsf{Counter}, \{\texttt{new} \ = \ \texttt{counter.new}, \\
&\quad \texttt{read} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{iseven}(\texttt{counter.get c}), \\
&\quad \texttt{toggle} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{counter.inc c}, \\
&\quad \texttt{reset} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{counter.new}\}\rangle \\
&\texttt{as}\ \exists \mathsf{FlipFlop}.\{\texttt{new} \ : \ \texttt{FlipFlop}, \\
&\quad \mathsf{read} \ : \ \mathsf{FlipFlop} \longrightarrow \mathsf{Bool}, \\
&\quad \mathsf{toggle} \ : \ \mathsf{FlipFlop} \longrightarrow \mathsf{FlipFlop}, \\
&\quad \mathsf{reset} \ : \ \mathsf{FlipFlop} \longrightarrow \mathsf{FlipFlop}\}
\end{aligned}
$$

Flip-flop

$$
\begin{aligned}
&\texttt{open}\ \langle \mathsf{Counter}, \texttt{counter}\rangle = \texttt{CounterADT in}\\
&\texttt{FlipFlopADT} =\\
&\quad \langle \mathsf{Counter}, \{\texttt{new} \ = \ \texttt{counter.new},\\
&\quad \texttt{read} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{iseven}(\texttt{counter.get c}),\\
&\quad \texttt{toggle} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{counter.inc c},\\
&\quad \texttt{reset} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{counter.new}\}\rangle\\
&\mathsf{as}\ \exists \mathsf{FlipFlop}.\{\texttt{new} \ : \ \texttt{FlipFlop},\\
&\quad \mathsf{read} \ : \ \mathsf{FlipFlop} \to \mathsf{Bool},\\
&\quad \mathsf{toggle} \ : \ \mathsf{FlipFlop} \to \mathsf{FlipFlop},\\
&\quad \mathsf{reset} \ : \ \mathsf{FlipFlop} \to \mathsf{FlipFlop}\}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{open}\ \langle \mathsf{FlipFlop}, \texttt{flipflop}\rangle = \texttt{FlipFlopADT in}\\
&\texttt{flipflop.read}\ (\texttt{flipflop.toggle}\ (\texttt{flipflop.toggle flipflop.new}));
\end{aligned}
$$

Flip-flop

$$
\begin{aligned}
&\texttt{open}\ \langle \mathsf{Counter}, \texttt{counter}\rangle = \texttt{CounterADT in}\\
&\texttt{FlipFlopADT} =\\
&\quad \langle \mathsf{Counter}, \{\texttt{new} \ = \ \texttt{counter.new},\\
&\quad \texttt{read} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{iseven}(\texttt{counter.get c}),\\
&\quad \texttt{toggle} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{counter.inc c},\\
&\quad \texttt{reset} \ = \ \lambda \texttt{c} : \mathsf{Counter}.\ \texttt{counter.new}\}\rangle\\
&\mathsf{as}\ \exists \mathsf{FlipFlop}.\{\texttt{new} \ : \ \texttt{FlipFlop},\\
&\quad \mathsf{read} \ : \ \mathsf{FlipFlop} \to \mathsf{Bool},\\
&\quad \mathsf{toggle} \ : \ \mathsf{FlipFlop} \to \mathsf{FlipFlop},\\
&\quad \mathsf{reset} \ : \ \mathsf{FlipFlop} \to \mathsf{FlipFlop}\}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{open}\ \langle \mathsf{FlipFlop}, \texttt{flipflop}\rangle = \texttt{FlipFlopADT in}\\
&\texttt{flipflop.read}\ (\texttt{flipflop.toggle}\ (\texttt{flipflop.toggle flipflop.new}));\\
&\triangleright\ \texttt{false} : \mathsf{Bool}
\end{aligned}
$$

Representation independence

- alternative implementation of the CounterADT:

$$
\begin{aligned}
&\texttt{CounterADT} = \\
&\quad \langle \{\mathsf{x} : \mathsf{Int}\}, \{ \\
&\quad \texttt{new} \;=\; \{\texttt{x} = 1\}, \\
&\quad \texttt{get} \;=\; \lambda\texttt{n} : \{\mathsf{x} : \mathsf{Int}\}.\ \texttt{n.x} \\
&\quad \texttt{inc} \;=\; \lambda\texttt{n} : \{\mathsf{x} : \mathsf{Int}\}.\ \{\texttt{x} = \texttt{succ}(\texttt{n.x})\}\}\rangle \\
&\texttt{as}\ \exists\mathsf{Counter}.\{ \\
&\quad \mathsf{new} \;:\; \mathsf{Counter}, \\
&\quad \mathsf{get} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Int}, \\
&\quad \mathsf{inc} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Counter}\}
\end{aligned}
$$

Representation independence

- alternative implementation of the CounterADT:

$$
\begin{aligned}
&\mathtt{CounterADT} = \\
&\quad \langle \{\mathsf{x} : \mathsf{Int}\}, \{ \\
&\quad\quad \mathtt{new} \;=\; \{\mathtt{x} = 1\}, \\
&\quad\quad \mathtt{get} \;=\; \lambda \mathtt{n} : \{\mathsf{x} : \mathsf{Int}\}.\ \mathtt{n.x} \\
&\quad\quad \mathtt{inc} \;=\; \lambda \mathtt{n} : \{\mathsf{x} : \mathsf{Int}\}.\ \{\mathtt{x} = \mathtt{succ}(\mathtt{n.x})\}\}\rangle \\
&\mathtt{as}\ \exists \mathsf{Counter}.\{ \\
&\quad\quad \mathsf{new} \;:\; \mathsf{Counter}, \\
&\quad\quad \mathsf{get} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Int}, \\
&\quad\quad \mathsf{inc} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Counter}\}
\end{aligned}
$$

- *representation independence*: follows from parametricity: the whole program remains typesafe since the counter instances cannot be accessed except using ADT operations

Representation independence

- alternative implementation of the CounterADT:

$$
\begin{aligned}
&\mathtt{CounterADT} = \\
&\quad \langle \{\mathsf{x} : \mathsf{Int}\}, \{ \\
&\quad \mathtt{new} \;=\; \{\mathtt{x} = \mathtt{1}\}, \\
&\quad \mathtt{get} \;=\; \lambda\mathtt{n} : \{\mathsf{x} : \mathsf{Int}\}.\ \mathtt{n.x} \\
&\quad \mathtt{inc} \;=\; \lambda\mathtt{n} : \{\mathsf{x} : \mathsf{Int}\}.\ \{\mathtt{x} = \mathtt{succ}(\mathtt{n.x})\}\}\rangle \\
&\mathtt{as}\ \exists\mathsf{Counter}.\{ \\
&\quad \mathsf{new} \;:\; \mathsf{Counter}, \\
&\quad \mathsf{get} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Int}, \\
&\quad \mathsf{inc} \;:\; \mathsf{Counter} \longrightarrow \mathsf{Counter}\}
\end{aligned}
$$

- *representation independence*: follows from parametricity: the whole program remains typesafe since the counter instances cannot be accessed except using ADT operations
- Mitchell 1991, Pitts 98

ADT-style of programming

- yields huge improvements in robustness and maintainability of large systems:

ADT-style of programming

- yields huge improvements in robustness and maintainability of large systems:
  - limits the scope of changes to the program

ADT-style of programming

- yields huge improvements in robustness and maintainability of large systems:
  - limits the scope of changes to the program
  - encourages the programmer to limit the dependencies between parts of the program (by making the signatures of the ADTs as small as possible)

ADT-style of programming

- yields huge improvements in robustness and maintainability of large systems:
  - limits the scope of changes to the program
  - encourages the programmer to limit the dependencies between parts of the program (by making the signatures of the ADTs as small as possible)
  - forces programmers to think about designing abstractions

Overview

1. Typing and evaluation rules
2. Encoding existential types
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) **Introducing objects**
   (c) Objects vs. ADTs

Existential objects

- two basic components: internal state, methods to manipulate the state:

$$
\begin{aligned}
\mathtt{c} \quad = \quad & \langle \mathsf{Int}, \{ \\
& \quad \mathtt{state} = 5, \\
& \quad \mathtt{methods} = \{ \\
& \qquad \mathtt{get} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{x}, \\
& \qquad \mathtt{inc} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\} \\
\mathtt{as} \quad & \exists \mathsf{X}. \{ \\
& \quad \mathsf{state} : \mathsf{X}, \\
& \quad \mathsf{methods} : \{ \\
& \qquad \mathsf{get} : \mathsf{X} \longrightarrow \mathsf{Int}, \\
& \qquad \mathsf{inc} : \mathsf{X} \longrightarrow \mathsf{X}\}\}
\end{aligned}
$$

Invoking the get method

$$
\begin{aligned}
\mathtt{c} \quad = \quad & \langle \mathsf{Int}, \{ \\
& \quad \mathtt{state} \;=\; 5, \\
& \quad \mathtt{methods} \;=\; \{ \\
& \qquad \mathtt{get} \;=\; \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{x}, \\
& \qquad \mathtt{inc} \;=\; \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\} \\
\mathtt{as} \quad & \exists \mathsf{X}.\{ \\
& \quad \mathsf{state} \;:\; \mathsf{X}, \\
& \quad \mathsf{methods} \;:\; \{ \\
& \qquad \mathsf{get} \;:\; \mathsf{X} \longrightarrow \mathsf{Int}, \\
& \qquad \mathsf{inc} \;:\; \mathsf{X} \longrightarrow \mathsf{X}\}\}
\end{aligned}
$$

$$
\begin{aligned}
& \mathtt{open}\ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c}\ \mathtt{in} \\
& \mathtt{body.methods.get}\ (\mathtt{body.state});
\end{aligned}
$$

Invoking the get method

$$
\begin{aligned}
\mathtt{c} \quad = \quad & \langle \mathsf{Int}, \{ \\
& \quad \mathtt{state} \; = \; 5, \\
& \quad \mathtt{methods} \; = \; \{ \\
& \qquad \mathtt{get} \; = \; \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{x}, \\
& \qquad \mathtt{inc} \; = \; \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\} \\
\mathtt{as} \quad & \exists \mathsf{X}.\{ \\
& \quad \mathsf{state} \; : \; \mathsf{X}, \\
& \quad \mathsf{methods} \; : \; \{ \\
& \qquad \mathsf{get} \; : \; \mathsf{X} \longrightarrow \mathsf{Int}, \\
& \qquad \mathsf{inc} \; : \; \mathsf{X} \longrightarrow \mathsf{X}\}\}
\end{aligned}
$$

$\mathtt{open}\ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c}\ \mathtt{in}$

$\mathtt{body.methods.get}\ (\mathtt{body.state});$

$\triangleright\ 5 : \mathsf{Int}$

Encapsulating the get method

$$
\begin{aligned}
\mathsf{C} \quad &= \quad \exists \mathsf{X}.\{ \\
&\qquad \mathsf{state} \ : \ \mathsf{X}, \\
&\qquad \mathsf{methods} \ : \ \{ \\
&\qquad\quad \mathsf{get} \ : \ \mathsf{X} \rightarrow \mathsf{Int}, \\
&\qquad\quad \mathsf{inc} \ : \ \mathsf{X} \rightarrow \mathsf{X}\}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{sendget} \quad &= \quad \lambda \mathtt{c} : \mathsf{C}. \\
&\qquad \mathtt{open}\ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c}\ \mathtt{in} \\
&\qquad \mathtt{body.methods.get}\ (\mathtt{body.state})
\end{aligned}
$$

Invoking the inc method

$$
\begin{aligned}
\mathtt{c} \quad = \quad & \langle \mathsf{Int}, \{ \\
& \quad \mathtt{state} \ = \ 5, \\
& \quad \mathtt{methods} \ = \ \{ \\
& \qquad \mathtt{get} \ = \ \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{x}, \\
& \qquad \mathtt{inc} \ = \ \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\} \\
\mathtt{as} \quad & \exists \mathsf{X}.\{ \\
& \quad \mathsf{state} \ : \ \mathsf{X}, \\
& \quad \mathsf{methods} \ : \ \{ \\
& \qquad \mathsf{get} \ : \ \mathsf{X} \rightarrow \mathsf{Int}, \\
& \qquad \mathsf{inc} \ : \ \mathsf{X} \rightarrow \mathsf{X}\}\}
\end{aligned}
$$

$$
\begin{aligned}
& \mathtt{open} \ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c} \ \mathtt{in} \\
& \mathtt{body.methods.inc} \ (\mathtt{body.state});
\end{aligned}
$$

Invoking the inc method

$$
\begin{array}{lll}
\mathtt{c} & = & \langle \mathsf{Int}, \{ \\
& & \quad \mathtt{state} \ = \ 5, \\
& & \quad \mathtt{methods} \ = \ \{ \\
& & \qquad \mathtt{get} \ = \ \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{x}, \\
& & \qquad \mathtt{inc} \ = \ \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x})\}\} \\
& \mathtt{as} & \exists \mathsf{X}.\{ \\
& & \quad \mathsf{state} \ : \ \mathsf{X}, \\
& & \quad \mathsf{methods} \ : \ \{ \\
& & \qquad \mathsf{get} \ : \ \mathsf{X} \to \mathsf{Int}, \\
& & \qquad \mathsf{inc} \ : \ \mathsf{X} \to \mathsf{X}\}\}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{open}\ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c}\ \mathtt{in} \\
\mathtt{body.methods.inc}\ (\mathtt{body.state}); \\
\triangleright\ \mathtt{Error} :\ \mathtt{scoping\ error}
\end{array}
$$

Invoking the inc method

$$
\begin{aligned}
\mathtt{c} \quad = \quad & \langle \mathsf{Int}, \{ \\
& \quad \mathtt{state} = 5, \\
& \quad \mathtt{methods} = \{ \\
& \qquad \mathtt{get} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{x}, \\
& \qquad \mathtt{inc} = \lambda \mathtt{x} : \mathsf{Int}.\ \mathtt{succ}(\mathtt{x}) \} \} \\
\mathtt{as} \quad & \exists \mathsf{X}. \{ \\
& \quad \mathsf{state} : \mathsf{X}, \\
& \quad \mathsf{methods} : \{ \\
& \qquad \mathsf{get} : \mathsf{X} \rightarrow \mathsf{Int}, \\
& \qquad \mathsf{inc} : \mathsf{X} \rightarrow \mathsf{X} \} \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathtt{open}\ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c}\ \mathtt{in} \\
& \mathtt{body.methods.inc}\ (\mathtt{body.state}); \\
& \triangleright\ \mathtt{Error:}\ \ \mathtt{scoping\ error}
\end{aligned}
$$

- why? X appears free in the body of the `open`

Encapsulating the inc method

- in order to properly invoke the `inc` method, we must repackage the fresh internal state as a counter object:

$$
\begin{aligned}
\mathsf{C} \quad = \quad & \exists \mathsf{X}.\{ \\
& \quad \mathsf{state} : \mathsf{X}, \\
& \quad \mathsf{methods} : \{ \\
& \qquad \mathsf{get} : \mathsf{X} \to \mathsf{Int}, \\
& \qquad \mathsf{inc} : \mathsf{X} \to \mathsf{X}\}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{sendinc} \quad = \quad & \lambda \mathtt{c} : \mathsf{C}. \\
& \quad \mathtt{open}\ \langle \mathsf{X}, \mathtt{body} \rangle = \mathtt{c}\ \mathtt{in} \\
& \qquad \langle \mathsf{X}, \{ \\
& \qquad\quad \mathtt{state} = \mathtt{body.methods.inc}\ (\mathtt{body.state}), \\
& \qquad\quad \mathtt{methods} = \mathtt{body.methods}\} \rangle \\
& \qquad \mathtt{as}\ \mathsf{C};
\end{aligned}
$$

Overview

1. Typing and evaluation rules
2. Encoding existential types
3. Abstract data types (ADTs) and objects
   (a) Introducing ADTs
   (b) Introducing objects
   (c) **Objects vs. ADTs**

Abstract type of counters

- ADT-style: counter values are elements of the underlying representation (i.e. simple numbers of type `Int`)

Abstract type of counters

- ADT-style: counter values are elements of the underlying representation (i.e. simple numbers of type Int)
- object-style: each counter is a whole module, including not only the internal representation but also the methods. Type $\mathsf{Counter}$ stands for the whole existential type:

$$
\begin{aligned}
&\exists \mathsf{X}.\{\\
&\quad \mathsf{state} \;:\; \mathsf{X},\\
&\quad \mathsf{methods} \;:\; \{\\
&\qquad \mathsf{get} \;:\; \mathsf{X} \to \mathsf{Int},\\
&\qquad \mathsf{inc} \;:\; \mathsf{X} \to \mathsf{X}\}\}
\end{aligned}
$$

Stylistic advantages

- advantage of the object-style: since each object chooses its own representation and operations, different implementations of the same object can be freely intermixed

Stylistic advantages

- advantage of the object-style: since each object chooses its own representation and operations, different implementations of the same object can be freely intermixed
- advantage of the ADT-style: binary operations (i.e. operations that accept ≥ 2 arguments of the abstract type) can be implemented, contrary to objects

Stylistic advantages

- advantage of the object-style: since each object chooses its own representation and operations, different implementations of the same object can be freely intermixed
- advantage of the ADT-style: binary operations (i.e. operations that accept ≥ 2 arguments of the abstract type) can be implemented, contrary to objects

Binary operations and the object-style

- e.g. set objects type:

$$
\begin{aligned}
\mathsf{IntSet} = \{\exists X,\ \{\mathsf{state} : \ X, \mathsf{methods} : \{&\mathsf{empty} : X, \\
&\mathsf{singleton} : \mathsf{Int} \longrightarrow X, \\
&\mathsf{member} : X \longrightarrow \mathsf{Int} \longrightarrow \mathsf{Bool}, \\
&\mathsf{union} : X \longrightarrow \mathsf{IntSet} \longrightarrow X\}\}
\end{aligned}
$$

Binary operations and the object-style

- e.g. set objects type:

$$\begin{aligned}\mathsf{IntSet} = \{\exists X,\ \{\mathsf{state} :\ X, \mathsf{methods} : \{&\mathsf{empty} : X,\\ &\mathsf{singleton} : \mathsf{Int} \to X,\\ &\mathsf{member} : X \to \mathsf{Int} \to \mathsf{Bool},\\ &\mathsf{union} : X \to \mathsf{IntSet} \to X\}\}\end{aligned}$$

- cannot implement the method since it can have no access to the concrete representation of the second argument

Binary operations and the object-style

- e.g. set objects type:

$$\begin{aligned} \mathsf{IntSet} = \{\exists \mathsf{X},\ \{\mathsf{state}:\ \mathsf{X}, \mathsf{methods}: \{&\mathsf{empty}: \mathsf{X}, \\ &\mathsf{singleton}: \mathsf{Int} \to \mathsf{X}, \\ &\mathsf{member}: \mathsf{X} \to \mathsf{Int} \to \mathsf{Bool}, \\ &\mathsf{union}: \mathsf{X} \to \mathsf{IntSet} \to \mathsf{X}\}\} \end{aligned}$$

- cannot implement the method since it can have no access to the concrete representation of the second argument
- in reality, mainstream OO languages such as C++ and Java have a hybrid object model that allows binary operations (with the the cost of restricting type equivalence)

Summary

- existential types are another form of polymorphism

Summary

- existential types are another form of polymorphism
- existentials can be encoded using universal types

Summary

- existential types are another form of polymorphism
- existentials can be encoded using universal types
- parametricity of existentials leads to representation independence

Summary

- existential types are another form of polymorphism
- existentials can be encoded using universal types
- parametricity of existentials leads to representation independence
- trade-offs between ADTs and objects:

Summary

- existential types are another form of polymorphism
- existentials can be encoded using universal types
- parametricity of existentials leads to representation independence
- trade-offs between ADTs and objects:
    - ADTs support binary operations, objects do not

Summary

- existential types are another form of polymorphism
- existentials can be encoded using universal types
- parametricity of existentials leads to representation independence
- trade-offs between ADTs and objects:
  - ADTs support binary operations, objects do not
  - objects support free intermixing of different implementations, ADTs do not

References

- Benjamin C. Pierce 2002: “Types and Programming Languages”
- Andrew M. Pitts 1998: “Existential Types: Logical Relations and Operational Equivalence”
- John C. Mitchell 1991: “On the Equivalence of Data Representations”
- Luca Cardelli and Xavier Leroy: “Abstract types and the dot notation”