Existential Types and ADTs

Christine Rizkallah
CSE, UNSW
Term 3 2020

Motivation

Throughout your studies, lecturers have (hopefully) expounded on the software engineering advantages of *abstract data types*.

So what is an abstra

Definition

An *abstract data type* is a type defined not by its internal re
operations that can be performed on it.
Typically these operations are specified using a more abstrac
implementation.

Language Examples: C

How do we do it in C?

stack.h

```
typedef stack_impl *Stack;

Stack empty(
Stack push(S
Stack pop(St
bool isEmpty(Stack);
void destroy(Stack);
```

```
struct stack_im
    int head;
    Stack tail;
}

Stack empty() { ... }
...
```

By only importing stack.h, we hide the implementation.

Language Examples: Haskell

Define a module but restrict what is exported:


```
module Stack
    ( Stack -- Cons and Nil are *not* exported
    , empty
    , push
    , pop
    , isEmpty
    ) where

    data Stack = Cons Int Stack | Nil

    empty :: Stack
    empty = Nil
    ...
```

Language Examples: Java

Typically Java accomplishes this with subtype polymorphism, something we discuss in the next lecture.


```
public interface Stack {
    public vo
    public in
    public bo
}

public class ListStack implements Stack {
    public ListStack() { ... };

    ...
}
```

Language Examples: Python

No luck here.

Quote

"Python is very sim
down the road, if yo
That's not good because big programs require modularity and encapsulation and you'd like a language that could support that."
Barbara Liskov, *The Power of Abstraction*, 2013.

You don't need static types to enforce abstraction, but it helps.

MinHS

How can we support abstract data types in MinHS? Can we use existing features to do so? We can use parametric polymorphism:

(**type** .

::

$\forall \mathcal{S}.\ (\mathcal{S} \to \texttt{Int} \to \mathcal{S})$ (push)
$\to (\mathcal{S} \to \mathcal{S} \times \texttt{Int})$ (pop)
$\to (\mathcal{S} \to \texttt{Bool})$ (isEmpty)
$\to \mathcal{S}$ (empty)
$\to \texttt{Bool}$

The program *foo* is defined for any stack type $\mathcal{S}$. Implementations of the operations must be provided as parameters.

Modules

We would like a single value to pass around, that contains all the implementations of the stack interface. It's too cumbersome to pass around each function implementation individually like b

Our toy *foo* pr

StackModule

For some type StackModule. Taking in a value of type LE is analogous to importing the module.

Via Curry-Howard

Let's translate the type of *foo* into a proposition, then do logical transformations to it:

Perhaps do this on the whiteboard.

$$\forall \mathcal{S}.\ ((\mathcal{S} \to \texttt{Int} \to \mathcal{S}) \to (\mathcal{S} \to \mathcal{S} \times \texttt{Int}) \to (\mathcal{S} \to \texttt{Bool}) \to \mathcal{S} \to \texttt{Bool})$$

$$\forall \mathcal{S}.\ ((\mathcal{S} \Rightarrow \texttt{Int} \Rightarrow \mathcal{S}) \Rightarrow (\mathcal{S} \Rightarrow \mathcal{S} \wedge \texttt{Int}) \Rightarrow (\mathcal{S} \Rightarrow \texttt{Bool}) \Rightarrow \mathcal{S} \Rightarrow \texttt{Bool})$$

(as $P \Rightarrow Q \Rightarrow R = P \wedge Q \Rightarrow R$)

$$\forall \mathcal{S}.\ ((\mathcal{S} \Rightarrow \texttt{Int} \Rightarrow \mathcal{S}) \wedge (\mathcal{S} \Rightarrow \mathcal{S} \wedge \texttt{Int}) \wedge (\mathcal{S} \Rightarrow$$

(as $\forall X.(P(X) \Rightarrow Q) = (\exists X.P(X)$)

$$(\exists \mathcal{S}.\ (\mathcal{S} \Rightarrow \texttt{Int} \Rightarrow \mathcal{S}) \wedge (\mathcal{S} \Rightarrow \mathcal{S} \wedge \texttt{Int}) \wedge (\mathcal{S} \Rightarrow \texttt{Bool}) \wedge \mathcal{S}) \Rightarrow \texttt{Bool}$$

(back to types)

$$(\exists \mathcal{S}.\ (\mathcal{S} \to \texttt{Int} \to \mathcal{S}) \times (\mathcal{S} \to \mathcal{S} \times \texttt{Int}) \times (\mathcal{S} \to \texttt{Bool}) \times \mathcal{S}) \to \texttt{Bool}$$

Existential Types

We have our StackModule type:

$$(\exists S.\ (() \to S) \times (S \to S \times \texttt{Int}) \times (S \to \texttt{Bool}) \times (S \to \texttt{Bool} \times S)) \to \texttt{Bool}$$

But what is this

Existential vs Un

$\forall a.\ \tau$ When producing a value, a is an arbitrar
When consuming a value, a may be inst
desired type.

$\exists a.\ \tau$ When consuming a value, a is an arbitrary, unknown type.
When producing a value, a may be instantiated to any desired type.

Another, Smaller Example

An ADT Bag is specified by three operations:

1. *emptyBag*, which gives a new, empty bag.
2. *addToBa*
3. *average*, w

What's the type fo

$$\text{BAGMODULE} = \exists \beta.\ \underbrace{\beta}_{emptyBag} \times (\underbrace{\beta}_{a} \to \underbrace{\beta}_{a} \to \underbrace{\beta}_{a}) \times (\underbrace{\beta}_{a} \to \underbrace{\text{Int}}_{\text{Int}})$$

The type of a module is called its *signature*.

Making a Module

We can make a value of an existential type using the `Pack` expression.

$$\frac{\Delta \vdash \tau \ \mathbf{ok} \qquad \Delta; \Gamma \vdash e : \rho[a := \tau]}{\Delta; \Gamma \vdash \mathtt{Pack}\ (\tau, e)\ \mathtt{as}\ \exists a.\ \rho : \exists a.\ \rho}$$

Just as the type
$\exists a.\ \tau$ could be vie

Example (Bag as two integers)


```
Pack (Int × Int)
     (  (0, 0)
     ,  recfun addToBag b i = (fst b + i, snd b + 1)
     ,  recfun average b = (fst b ÷ snd b)
     )  :: BagModule
```

Importing a Module

If we are given a module as a parameter, we can access its contents using the `Open` expression:

$$\frac{\Delta;\Gamma \vdash e_1 : \exists a.\ \tau \qquad (\Delta, a\ \textbf{bound});(\Gamma, x:\tau) \vdash e_2 : \rho}{\quad}$$

The last two prem… …ype that is only in scope inside e_2.

Example (Averaging some numbers with a bag)


```
recfun example :: (BAGMODUL
  Open bagM
    (B. (empty, addToBag, average).
       average (addToBag (addToBag empty 60) 30)
    )
```

In Practice

Generally, most programming languages have fairly poor support for modules.

- Dynamica [1].
- Haskell wi
- Java and sim al typing in its full generality.
- Languages in the ML family, like SML and OCaml, have ve modules.

[1] What they call "modules" aren't. Just like types.