Existential Types and Abstraction

Johannes Åman Pohjola
UNSW

Term 3 2022

Motivation

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

So what is an abstract data type?

Motivation

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

So what is an abstract data type?

Definition

An *abstract data type* is a type defined not by its internal representation, but by the operations that can be performed on it.

Typically, these operations are specified using a more abstract model than the actual implementation.

Language Examples: C

How do we do it in C?

Language Examples: C

How do we do it in C?

stack.h

```c
typedef stack_impl *Stack;

Stack empty();
Stack push(Stack, int);
Stack pop(Stack, int*);
bool isEmpty(Stack);
void destroy(Stack);
```

Language Examples: C

How do we do it in C?

stack.h

```c
typedef stack_impl *Stack;

Stack empty();
Stack push(Stack, int);
Stack pop(Stack, int*);
bool isEmpty(Stack);
void destroy(Stack);
```

stack.c

```c
#include "stack.h"

struct stack_impl {
    int head;
    Stack tail;
}

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

Language Examples: C

How do we do it in C?

stack.h

```
typedef stack_impl *Stack;

Stack empty();
Stack push(Stack, int);
Stack pop(Stack, int*);
bool isEmpty(Stack);
void destroy(Stack);
```

stack.c

```
#include "stack.h"

struct stack_impl {
    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 void push(int x);
    public int pop() throws EmptyStackException;
    public boolean isEmpty();
}

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

    ...
}
```

Language Examples: Python

Language Examples: Python

No luck here.

Quote

"Python is very simple and nice when you start to use it, but you don't get too far down the road, if you're me, before you discover it has no data abstraction at all. 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?

MinHS

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

$$
\begin{array}{l}
(\textbf{type}\ \mathcal{S}.\\
\quad \textbf{recfun}\ \mathit{foo}\ \mathit{push}\ \mathit{pop}\ \mathit{isEmpty}\ \mathit{empty} =\\
\quad\quad \textbf{let}\ s = \mathit{push}\ \mathit{empty}\ 42\\
\quad\quad \textbf{in}\ \mathit{isEmpty}\ (\mathsf{fst}\ (\mathit{pop}\ s)))\\
\quad\quad\quad\quad\quad\quad ::
\end{array}
$$

MinHS

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

$$
\begin{array}{l}
(\mathbf{type}\ \mathcal{S}.\\
\quad \mathbf{recfun}\ \mathit{foo}\ \mathit{push}\ \mathit{pop}\ \mathit{isEmpty}\ \mathit{empty} =\\
\quad\quad \mathbf{let}\ s = \mathit{push}\ \mathit{empty}\ 42\\
\quad\quad \mathbf{in}\ \mathit{isEmpty}\ (\mathsf{fst}\ (\mathit{pop}\ s)))
\end{array}
$$

$$::$$

$$
\begin{array}{ll}
\forall \mathcal{S}.\ (\mathcal{S} \to \mathtt{Int} \to \mathcal{S}) & (\text{push})\\
\ \to (\mathcal{S} \to \mathcal{S} \times \mathtt{Int}) & (\text{pop})\\
\ \to (\mathcal{S} \to \mathtt{Bool}) & (\text{isEmpty})\\
\ \to \mathcal{S} & (\text{empty})\\
\ \to \mathtt{Bool} &
\end{array}
$$

MinHS

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

$$
\begin{array}{l}
(\textbf{type}\ \mathcal{S}.\\
\quad \textbf{recfun}\ \mathit{foo}\ \mathit{push}\ \mathit{pop}\ \mathit{isEmpty}\ \mathit{empty} =\\
\quad\quad \textbf{let}\ s = \mathit{push}\ \mathit{empty}\ 42\\
\quad\quad \textbf{in}\ \mathit{isEmpty}\ (\mathsf{fst}\ (\mathit{pop}\ s)))
\end{array}
$$

$$::$$

$$
\begin{array}{ll}
\forall \mathcal{S}.\ (\mathcal{S} \rightarrow \mathtt{Int} \rightarrow \mathcal{S}) & (\mathsf{push})\\
\quad \rightarrow (\mathcal{S} \rightarrow \mathcal{S} \times \mathtt{Int}) & (\mathsf{pop})\\
\quad \rightarrow (\mathcal{S} \rightarrow \mathtt{Bool}) & (\mathsf{isEmpty})\\
\quad \rightarrow \mathcal{S} & (\mathsf{empty})\\
\quad \rightarrow \mathtt{Bool} &
\end{array}
$$

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 the whole stack interface. It's too cumbersome to pass around each component individually like before. This value is called a *module*.

Modules

We would like a single value to pass around, that contains the whole stack interface. It's too cumbersome to pass around each component individually like before. This value is called a *module*.

Our toy *foo* program from earlier needs to be rewritten as:

$$\textsc{StackModule} \rightarrow \texttt{Bool}$$

For some type $\textsc{StackModule}$. Taking in a value of type $\textsc{StackModule}$ 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.

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

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.

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

(translating to logic)

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.

$$\forall \mathcal{S}.\ ((\mathcal{S} \rightarrow \mathtt{Int} \rightarrow \mathcal{S}) \rightarrow (\mathcal{S} \rightarrow \mathcal{S} \times \mathtt{Int}) \rightarrow (\mathcal{S} \rightarrow \mathtt{Bool}) \rightarrow \mathcal{S} \rightarrow \mathtt{Bool})$$

(translating to logic)

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

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.

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

(translating to logic)

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

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

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.

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

(translating to logic)

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

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

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

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.

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

(translating to logic)

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

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

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

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

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.

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

(translating to logic)

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

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

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

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

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

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.

$$\forall \mathcal{S}.\ ((\mathcal{S} \rightarrow \mathtt{Int} \rightarrow \mathcal{S}) \rightarrow (\mathcal{S} \rightarrow \mathcal{S} \times \mathtt{Int}) \rightarrow (\mathcal{S} \rightarrow \mathtt{Bool}) \rightarrow \mathcal{S} \rightarrow \mathtt{Bool})$$

(translating to logic)

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

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

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

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

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

(back to types)

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.

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

(translating to logic)

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

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

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

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

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

(back to types)

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

Existential Types

We have our StackModule type:

$$\underbrace{(\exists \mathcal{S}.\ (\mathcal{S} \rightarrow \texttt{Int} \rightarrow \mathcal{S}) \times (\mathcal{S} \rightarrow \mathcal{S} \times \texttt{Int}) \times (\mathcal{S} \rightarrow \texttt{Bool}) \times \mathcal{S})}_{\textsc{StackModule}} \rightarrow \texttt{Bool}$$

But what is this $\exists a.\ \tau$ thing?

Existential Types

We have our StackModule type:

$$\underbrace{(\exists \mathcal{S}.\ (\mathcal{S} \rightarrow \texttt{Int} \rightarrow \mathcal{S}) \times (\mathcal{S} \rightarrow \mathcal{S} \times \texttt{Int}) \times (\mathcal{S} \rightarrow \texttt{Bool}) \times \mathcal{S})}_{\textsc{StackModule}} \rightarrow \texttt{Bool}$$

But what is this $\exists a.\ \tau$ thing?

Existential vs Universal Types

$\forall a.\ \tau$	When **producing** a value, a is an arbitrary, unknown type. When **consuming** a value, a may be instantiated to any 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. *addToBag*, which adds an integer to the bag.
3. *average*, which gives the arithmetic mean of the bag.

What's the type for this?

Another, Smaller Example

An ADT `Bag` is specified by three operations:

1. *emptyBag*, which gives a new, empty bag.
2. *addToBag*, which adds an integer to the bag.
3. *average*, which gives the arithmetic mean of the bag.

What's the type for this?

$$\textsc{BagModule} \quad = \quad \exists \mathcal{B}.\ \underbrace{\mathcal{B}}_{emptyBag} \times \overbrace{(\mathcal{B} \rightarrow \mathtt{Int} \rightarrow \mathcal{B})}^{addToBag} \times \overbrace{(\mathcal{B} \rightarrow \mathtt{Int})}^{average}$$

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.

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

Just as the type $\forall a.\ \tau$ could be viewed as a function from a type to a value, the type $\exists a.\ \tau$ could be viewed as a pair of a type and a value.

Making a Module

We can make a value of an existential type using the $\texttt{Pack}$ expression.

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

Just as the type $\forall a.\ \tau$ could be viewed as a function from a type to a value, the type $\exists a.\ \tau$ could be viewed as a pair of a type and a value.

Example (Bag as two integers)

$$
\begin{array}{lll}
\texttt{Pack}\ (\texttt{Int} \times \texttt{Int}) & & \\
 & (& (0, 0) \\
 & , & \mathsf{recfun}\ \mathit{addToBag}\ b\ i = (\mathsf{fst}\ b + i, \mathsf{snd}\ b + 1) \\
 & , & \mathsf{recfun}\ \mathit{average}\ b = (\mathsf{fst}\ b \div \mathsf{snd}\ b) \\
 &) & :: \textsc{BagModule}
\end{array}
$$

Importing a Module

If we have a module, we can access its contents using `Open`:

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

The last two premises ensure that the type ρ does not contain the abstract type—it is only in scope inside e_2.

Importing a Module

If we have a module, we can access its contents using `Open`:

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

The last two premises ensure that the type ρ does not contain the abstract type—it is only in scope inside e_2.

Example (Averaging some numbers with a bag)

$$
\begin{array}{l}
\textbf{recfun}\ f :: (\textsc{BagModule} \rightarrow \texttt{Int})\ \mathit{bagM} = \\
\quad \texttt{Open}\ \mathit{bagM} \\
\qquad (\mathcal{B}.\ (\mathit{empty}, \mathit{addToBag}, \mathit{average}). \\
\qquad\quad \mathit{average}\ (\mathit{addToBag}\ (\mathit{addToBag}\ \mathit{empty}\ 60)\ 30) \\
\qquad)
\end{array}
$$

Type inference?

Full type inference for existential types is an open research problem.

$$
\begin{array}{l}
\textbf{recfun}\ f\ b = \\
\quad \textbf{if}\ b\ \textbf{then} \\
\quad\quad (1, \lambda y.\ y + 1) \\
\quad \textbf{else} \\
\quad\quad (\texttt{true}, \lambda y.\ 1)
\end{array}
$$

Q: What's the type of f?

Type inference?

Full type inference for existential types is an open research problem.

$$
\begin{array}{l}
\textbf{recfun}\ f\ b = \\
\quad \textbf{if}\ b\ \textbf{then} \\
\quad\quad (1, \lambda y.\ y + 1) \\
\quad \textbf{else} \\
\quad\quad (\texttt{true}, \lambda y.\ 1)
\end{array}
$$

Q: What's the type of f?

A: Either of these:

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

...but neither is more general.

Type inference?

Full type inference for existential types is an open research problem.

$$
\begin{array}{l}
\textbf{recfun}\ f\ b = \\
\quad \textbf{if}\ b\ \textbf{then} \\
\qquad (1, \lambda y.\ y + 1) \\
\quad \textbf{else} \\
\qquad (\mathtt{true}, \lambda y.\ 1)
\end{array}
$$

Q: What's the type of f?

A: Either of these:

$$\mathtt{Bool} \rightarrow \exists a.\ a \times (a \rightarrow \mathtt{Int})$$
$$\mathtt{Bool} \rightarrow \exists a.\ a \times (\mathtt{Int} \rightarrow \mathtt{Int})$$

...but neither is more general.

Algorithms do exist with additional restrictions or annotations. See e.g. Eisenberg et. al, ICFP 2021.

In Practice

Programming language support for modules is a mixed bag.

- Dynamically typed languages typically don't support them at all[1].
- Haskell without extensions, C, and Go have very weak support for them.
- Rust has a feature called *impl Traits* which are a limited form of existential types.
- Java and similar accomplish modularity via OOP, which don't support existential typing in its full generality.
- Languages in the ML family, like SML and OCaml have very good support for modules, but typically not modules-as-expressions.

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