Covered so far:

- Syntax of programming languages
- Semantics of programming languages:
  - static semantics: scopes, types
  - dynamic semantics: big step/small step semantics
- Abstract machines:
  - substitution
  - environments
  - control stacks
- Exceptions, Continuations
- Polymorphism
- Aggregate data types
- Type inference
- Overloading and some aspects of type classes

What’s next?

- Data abstraction
- Subtyping
- Featherweight Java
- Inheritance

Data abstraction is a static property
- checked at compile time
- most languages realise this via scoping; private methods/declarations are not visible outside of the class or module

The programming language ML realises data abstraction via its type system.

Advantages:

- more powerful, more flexible approach
- ADTs are “first class citizens”
Modelling Data Abstraction in MinHs

We extend MinHs to handle ML-style data abstraction

Example: Queues
Assume we want to implement an abstract data structure queue, which provide the following operations:
- \texttt{empty}: Queue
- \texttt{enq}: \texttt{(Int, Queue)} \rightarrow \texttt{Queue}
- \texttt{deq}: \texttt{Queue} \rightarrow \texttt{(Int, Queue)}

Different implementations possible
\rightarrow simple list implementation: inefficient
\rightarrow pair of lists: linear work complexity

We want to hide concrete implementation.

In ML, the interface of the ADT would be given by the following signature:

\begin{verbatim}
signature QUEUE
sig
  type queue
  val empty: queue
  val enq : int * queue \rightarrow queue
  val deq : queue \rightarrow int * queue
end
\end{verbatim}

and the implementation, separate from the signature:

\begin{verbatim}
structure Q := Queue
struct
  type q = ..... 
  val empty = ..... 
  val enq = ..... 
  ....
\end{verbatim}

We don’t introduce a full module system into MinHs, but offer a simple mechanism to specify the interface and bundle it with an implementation.

In (explicitly typed) MinHs:
\rightarrow a list of integer values: \texttt{rec IntList. ((,) + (Int, IntList))}
\rightarrow an empty list: \texttt{roll (inl())}
\rightarrow the enqueue function:
\texttt{enq: (Int, rec t. ((,) + (Int, t))) \rightarrow (rec t. () + (Int, t))}
\texttt{letfun enq ar =}
\texttt{  roll (inr args)}
\rightarrow the dequeue function:
\texttt{deq: (rec t. ((,) + (Int, t))) \rightarrow (Int, rec t. ((,) + (Int*t))}
\texttt{letfun deq args =}
\texttt{  is}
\texttt{  ....}

In MinHs, we bundle all the functions of the ADT to a tuple\(^9\), together with the actual implementation type

\begin{verbatim}
pack(rec t.((,) + (Int,t)).roll (inl ()) . letfun enq ... ) : ?
\end{verbatim}

\(\text{concrete type of ADT}\hspace{1cm}\text{functions on ADT}\)

What is the type of a \texttt{pack}-expression?
\rightarrow actual implementation type should not be visible outside
\rightarrow the type of each function (parametrized over the ADT) should, however, be visible!
\rightarrow for brevity, we assume that \(n\)-ary tuples are available

\(\exists Q, (Q, ((Int, Q) \rightarrow Q), (Q \rightarrow (Int, Q)))\)

\(^9\)for brevity, we assume that \(n\)-ary tuples are available
Slide 9

Example: if \( e \) is the list ADT discussed earlier:

\[
e = \text{pack}((\text{rec } t().()) + (\text{Int } t().)), \text{roll}(\text{inl}()), \text{letfun enq}(...))
\]

then

\[
\begin{align*}
t &= \text{rec } t().() + (\text{Int } t().) \\
x &= (\text{roll } (\text{inl}()), \text{letfun enq}(...)) \\
fst(x) &= \text{roll } (\text{inl}()) \text{ (empty list) etc}
\end{align*}
\]

Making the operations \( e \) of an ADT \( t \) available in \( e' \):

\[
\text{open}(e, t, x, e')
\]

\( \rightarrow \) binds \( t \) to the abstract type, and
\( \rightarrow x \) to the implementation, i.e., to the \( n \)-tuple of functions on the type

Slide 10

Replacing the implementation of the ADT: Assume we change the implementation of the ADT to be a pair of lists:

\[
\begin{align*}
t_2 &= (\text{rec } t().() + (\text{Int } t().), \text{rec } t().() + (\text{Int } t().)) \\
e_2 \ll br\&
- \text{empty} &= (\text{roll } (\text{inl}()), \text{roll } (\text{inl}())) \\
- \ldots
\end{align*}
\]

then

\[
\text{open(pack}(r_2, e_2), t.x.e') = \text{open}(\text{pack}(t, e), t.x.e')
\]

The user code of the ADT does not have to change at all when we replace one concrete implementation of the ADT by another implementation.

Slide 11

Types: We extend the types of MinHs

Polytypes \( \sigma \) : \( \forall t. \sigma \mid \exists t. \sigma \mid \tau \)

Monotypes \( \tau \) : \( t \mid \ldots \)

Expressions \( e \) : \( \text{pack}(\tau, e) : \sigma \mid \text{open}(e_1, t.x : \sigma, e_2) \)

Values \( v \) : \( \text{pack}(\tau, e) : \sigma \mid \ldots \)

Static Semantics — Typing Rules

\[
\frac{\Delta \vdash \tau \text{ ok} \quad \Delta \vdash \exists t. \sigma \text{ ok} \quad \Gamma, \Delta \vdash e : (\tau/t)\sigma}{\Gamma, \Delta \vdash \text{pack}(\tau, e) : \exists t. \sigma}
\]

\[
\frac{\Delta \vdash \tau \text{ ok} \quad \Gamma \cup \{x : \sigma\}, \Delta \cup \{t\} \vdash e_c : \tau_e \quad \Gamma, \Delta \vdash e_{abs} : \exists t. \sigma \quad \Gamma \notin \Delta}{\Gamma, \Delta \vdash \text{open}(e_{abs}, t.x : \sigma, e_c) : \tau_e}
\]

- \( \Delta \vdash \tau \text{ ok} \): the type \( \tau \) of the client \( e_c \) may not contain \( t \) (note that we have \( t \notin \Delta \))
- \( \Gamma \cup \{x : \sigma\}, \Delta \cup \{t\} \vdash e_c : \tau_e \): the type of the client code \( e_c \) is checked without knowing what the representation type of \( t \) is.
- \( \Gamma, \Delta \vdash e_{abs} : \exists t. \sigma \): \( t \) may occur free in \( \sigma \)
We said that the Haskell type

```haskell
data Color = Red | Green | Blue
```

can be modelled in MinHs as

```haskell
(() + (() + ()))
```

However, the following type

```haskell
data Result = Yes | No | Maybe
```

would be mapped to exactly the same MinHs type.
This means that, for example, \textit{Yes} and \textit{Red} would be
represented by exactly the same term, namely \textit{inl ()}
Can we use data abstraction to distinguish these two types in
MinHs?