1 Data Structures

So far, the set of data types we can handle in MinHs is very restricted: we only have two primitive types, boolean and integers, and functions over these. All general purpose programming languages, however, also provide methods to combine these types to product, sum and recursive types.

1.1 Products

Programming languages usually offer some way to bundle values of (possibly) different types together to a new type, for example structures in C or tuples in Haskell.

In MinHs, we try again to keep things as simple as possible, and only extend the language with binary pairs \((\tau_1 \times \tau_2)\) and nullary tuples \((\text{Unit})\). The nullary tuple is a strange type, because it contains exactly one element (and is for this reason also called \(\text{Unit-type}\)), and therefore seems to be pretty pointless. We will see later, however, that it does makes sense in combination with other data type constructors.

1.1.1 Concrete and abstract syntax

The concrete and abstract syntax for the construction of pairs is as follows:

\[
\begin{align*}
(e_1, e_2) & \quad \text{pair}(e_1, e_2) \\
() & \quad \text{unit}\end{align*}
\]

We also need functions to access the components of binary pairs (obviously none for nullary tuples)

\[
\begin{align*}
fst(e) & \quad \text{fst}(e) \\
\text{snd}(e) & \quad \text{snd}(e)
\end{align*}
\]

And the concrete and abstract syntax of the types:

\[
\begin{align*}
\tau_1 \times \tau_2 & \quad \text{Cross}(\tau_1, \tau_2) \\
\text{Unit} & \quad \text{Unit}
\end{align*}
\]

1.1.2 Example

We can use pairs now to define another version of the \(\text{div}\) function:

\[
\begin{align*}
\text{letfun div}::(\text{Int}, \text{Int}) \rightarrow \text{Int} \quad \text{(args)} = \\
& \quad \text{if} \ (\text{fst} \ \text{args}) \ > \ (\text{snd} \ \text{args}) \ \text{then} \\
& \quad \quad 0 \\
& \quad \text{else} \ \text{div} \ ((\text{fst} \ \text{args}) \ - \ (\text{snd} \ \text{args}), \ (\text{snd} \ \text{args}))
\end{align*}
\]
1.1.3 Static Semantics: typing rules

We have to add the typing rules for pairs to the set of inference rules describing the static semantics of MinHs:

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{pair}(e_1, e_2) : \text{cross}(\tau_1, \tau_2)
\]

\[
\Gamma \vdash e : \text{cross}(\tau_1, \tau_2) \\
\Gamma \vdash \text{fst}(e) : \tau_1 \\
\Gamma \vdash \text{snd}(e) : \tau_2
\]

\[
\Gamma \vdash \text{unitel} : \text{Unit}
\]

1.1.4 Dynamic Semantics

The SOS rules for the evaluation of expressions containing pairs are also straightforward:

\[
e_1 \mapsto_M e'_1 \\
\text{pair}(e_1, e_2) \mapsto_M \text{pair}(e'_1, e_2)
\]

\[
e_2 \mapsto_M e'_2 \\
\text{pair}(e_1, e_2) \mapsto_M \text{pair}(e_1, e'_2)
\]

\[
\text{fst}(e) \mapsto_M \text{fst}(e') \\
\text{snd}(e) \mapsto_M \text{snd}(e')
\]

\[
\text{fst}(\text{pair}(e_1, e_2)) \mapsto_M e_1 \\
\text{snd}(\text{pair}(e_1, e_2)) \mapsto_M e_2
\]

A pair of expressions is evaluated by first reducing the first expression to a value, then the second (just like for all other operations). A pair of values cannot be evaluated further.

1.2 Sum Types

The product of two types is a new type which contains an element of the first and the second type. The sum of two types is a new type which contains elements of either the first or the second type, corresponding in some way to C-type union types.

We only add binary sums to MinHs, as it is possible to express combinations of more than two types by nesting binary sums.

1.2.1 Concrete and abstract syntax

Types: The concrete and abstract syntax to denote a sum type is as follows:

\[
\tau_1 + \tau_2 \\
\text{sum}(\tau_1, \tau_2)
\]

Constructors: We have two constructors, \text{inl} and \text{inr}, for the binary sum type, both have to be annotated with the name of both types:

\[
\text{inl}(\tau_1, \tau_2, e_1) \\
\text{inr}(\tau_1, \tau_2, e_2)
\]

For example, the term \text{inl}(\text{Bool}, \text{Int}, \text{True}) has type \text{Bool + Int}, as has term \text{inr}(\text{Bool}, \text{Int}, 5).
Operations: we also add a new language construct, case-expressions, to MinHs:

\[
\begin{align*}
\text{case } e \text{ of } & \text{ inl}(x) \rightarrow e_1 \\
& \text{ inr}(y) \rightarrow e_2 \\
& \text{ case}(\tau_1, \tau_2, e, x.e_1, y.e_2)
\end{align*}
\]

1.2.2 Examples
The following definition is a simple example of a function which uses a case-expression to inspect its argument:

\[
\text{letfun foo:: (Int + Bool) -> Int (x) = case x of inl (i) -> i inr (b) -> if b then 1 else 0}
\]

The type Int + () is similar to the type Maybe Int in Haskell — it is either a integer value or “nothing”, that is, an element of unit type. We can rewrite integer division to return a integer value if the denominator is non-zero, () otherwise:

\[
\text{letfun div'':: (Int, Int) -> (Int + ()) pr = if (snd pr) == 0 then inr () else inl (div pr)}
\]

where div is the previously defined (partial) division function.

1.2.3 Static Semantics: Typing rules
The typing rules for case-expressions make sure that, similar to if-expressions, the type of both branches are identical:

\[
\begin{align*}
\Gamma \vdash e : \text{sum}(\tau_1, \tau_2) & \quad \Gamma \cup \{x : \tau_1\} \vdash e_1 : \tau \\
& \quad \Gamma \cup \{y : \tau_2\} \vdash e_2 : \tau \\
\Gamma \vdash \text{case}(\tau_1, \tau_2, e, x.e_1, y.e_2) : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{inl}(\tau_1, \tau_2, e_1) : \text{sum}(\tau_1, \tau_2) & \quad \Gamma \vdash \text{inr}(\tau_1, \tau_2, e_2) : \text{sum}(\tau_1, \tau_2)
\end{align*}
\]

Since the programmer has to provide the types, the rules are, again, straight forward.

1.2.4 Dynamic Semantics: Evaluation Rules
For the definition of the dynamic semantics, we omit the type annotations. Note that there is no rule to further evaluate an expression of the form \text{inl}(e) or \text{inr}(e) – they are already fully evaluated.

\[
\begin{align*}
\begin{array}{c}
e \mapsto_M e' \\
\text{inl}(e) \mapsto_M \text{inl}(e')
\end{array}
& \quad
\begin{array}{c}
e \mapsto_M e' \\
\text{inr}(e) \mapsto_M \text{inr}(e')
\end{array}
\end{align*}
\]

We can only access the “contents” of \text{inl} and \text{inr}-expressions using a case-expression. As for if-expressions, the evaluation of a case-expression only triggers the evaluation of the expression...
which determines the branching — the branches themselves are only evaluated on demand:

\[
\begin{align*}
\text{case}(e, x.e_1, y.e_2) & \mapsto \text{case}(e', x.e_1, y.e_2) \\
\text{case}(\text{inl}(v), x.e_1, y.e_2) & \mapsto \{v/x\}e_1 \\
\text{case}(\text{inr}(v), x.e_1, y.e_2) & \mapsto \{v/y\}e_2
\end{align*}
\]

1.3 Recursive Types

With products and sums, we still have no way to express a type like the following Haskell `IntList` type in MiniHs:

```haskell
data IntList = Nil | IList Int IntList
```

The problem here is the recursion, `IntList` appears on the right hand side of the definition. To be able to express a similar type, we need a constructor for recursive types:

```haskell
rec <typeName> is <type>
```

The name `<typeName>` can occur freely anywhere in `<type>`, and refers to the whole type expression `rec <typeName> is <type>`. So, for example, a list of integer values can in this notation be defined as follows:

```haskell
rec List is (Unit + (Int * List))
```

The name of the recursive type `List` appears again in the type term. This defines `List` to be either a item of unit type or a pair of an integer value and a rest list, just as the `IntList` definition did. As for sum types with `inl` and `inr`, we need constructors to map a value to a value of recursive type, and an inverse operation. The constructor is call `roll (<type>, <term>)`. So, for example, elements of the type `List` have the following form (annotated with the corresponding Haskell list expressions):

- `roll(inl ()) [ ]`
- `roll (inr (1, roll(inl ())))) [1]`
- `roll (inr(2, roll (inr (1, roll(inl ())))))) [2,1]`

Note the term for the 'empty list': it is not just `()`, since this would have type `Unit`, and `inl ()` has type `Unit + Int * List`, and `roll (inl ())` finally has the required type: `rec List is (Unit + (Int * List))`.

The `unroll` function “unpacks” a recursive type and has basically the inverse effect of `roll`. The following function definition demonstrate how `roll` and `unroll` are used:

```haskell
letfun sumElems: rec List is (Unit + (Int * List)) -> Int rxs =
  let xs = unroll rxs
  in
  case xs of
    inl () -> 0
    inr pr -> fst pr + (sumElems (snd pr))

letfun buildList: Int -> rec List is (Unit + (Int * List)) n =
  if n == 0
    then roll (inl ())
    else roll (inr (n, buildList (n-1)))
```

\footnote{We omitted the required type annotation here: strictly speaking, we should write `inl (Unit + Int * List, ())` and `roll (rec List is (Unit + (Int * List)), ...)`}

4
1.4 Static Semantics and Dynamic Semantics

The static semantics of the new language constructs is as follows:

\[
\Gamma \vdash e : \{\text{rec}(t,\tau)/t\}\tau \\
\Gamma \vdash \text{roll}(e) : \text{rec}(t,\tau)
\]

\[
\Gamma \vdash e : \text{rec}(t,\tau) \\
\Gamma \vdash \text{unroll}(e) : \{\text{rec}(t,\tau)/t\}\tau
\]

And finally the dynamic semantics rules:

\[
\text{roll}(e) \mapsto_{M} \text{roll}(e') \\
\text{unroll}(e) \mapsto_{M} \text{unroll}(e')
\]

\[
\text{roll}(\text{unroll}(e)) \mapsto_{M} e
\]