Substitution

Here are a few more examples related to records and unions when working on capture-avoiding substitution:

\[
\text{substitute}(N(10), "x", \text{Record}(\text{Map}("x" \rightarrow \text{Var}("x"), \ "y" \rightarrow \text{Var}("x"))))
\]
should return
\[
\text{Record}(\text{Map}("x" \rightarrow N(10), \ "y" \rightarrow N(10)))
\]

\[
\text{substitute}(N(10), "x", \text{Match}(\text{TInt}, \text{Inj}(...),
\text{Map}("c1" \rightarrow ("x", \text{Var}("x")),
\ "c2" \rightarrow ("y", \text{Binary}(\text{Plus}, \text{Var}("x"), \text{Var}("y")))))
\]
should return
\[
\text{Match}(\text{TInt}, \text{Inj}(...),
\text{Map}("c1" \rightarrow ("x", \text{Var}("x")), \ "c2" \rightarrow ("y", \text{Binary}(\text{Plus}, 10, \text{Var}("y"))))
\]

Although the abstract syntax for \text{Inj}(...) is not filled in, substitution should happen recursively in there. But note that the crucial part of the substituting into \text{Match} expressions is in the new bindings that it introduces. Notice that \text{Record} does not introduce new bindings, whereas \text{Match} does. Recall that \text{DefFun} also introduces a new binding.

Record Types

We have seen tuples used in Scala. In our extended Smalla language we define record types which are similar, but instead of ordering the components we access the different components using labels.

Suppose Ralph wants to store information about a flight he’s taking for Spring Break. He wants to store three pieces of information: the cost of the flight (an Int), whether the seating is first class (a Boolean), and the length of the flight in kilometers (an Int).

We could represent this information as a triple (a tuple with three elements) in Scala with type (Int, Boolean, Int)

\[
(500, \text{true}, 5000): (\text{Int}, \text{Boolean}, \text{Int})
\]

To access an element in the tuple we can either use pattern matching, or an accessor specifying which index we want to retrieve from the tuple. In the interpreter:

\[
\text{scala} > (500, \text{true}, 5000).\_2
\]

\[
\text{res0}: \text{Boolean} = \text{true}
\]
Alternatively we could represent this data as a record – a map from field names to expressions. We specify a record type by defining which field names map to expressions of which type. So using records we could represent this data as:

\[
\{ \text{cost} \rightarrow 500, \text{isFirstClass} \rightarrow \text{true}, \text{length} \rightarrow 5000 \} : \{ \text{cost} \rightarrow \text{Int}, \text{isFirstClass} \rightarrow \text{Boolean}, \text{length} \rightarrow \text{Int} \}
\]

Here instead of the component number we use the field name to access a component of the record. Using one of the axioms from the SMALLA small-step operational semantics:

\[
\{ \text{cost} \rightarrow 500, \text{isFirstClass} \rightarrow \text{true}, \text{length} \rightarrow 5000 \}.\text{isFirstClass} \rightarrow \text{true}
\]

The abstract syntax for accesses uses the \texttt{GetField} expression:

\[
\text{GetField} (\text{Record} (\text{Map} (\text{cost} \rightarrow \text{N}(500), \text{isFirstClass} \rightarrow \text{B}(\text{true}), \text{length} \rightarrow \text{N}(5000))), \text{"isFirstClass"})
\]

Question – How do SMALLA reduction rules work if the field we are trying to access is not mapped to a value?

Labeled Union Types We have used case classes and pattern matching extensively so far in the course, and seen that they can be very useful. We will implement something similar in SMALLA using two constructs:

\texttt{case class Inj(t: TUnion, c: String, e1: Expr) extends Expr}  
\texttt{case class Match(t: Typ, e1: Expr, cases: Map[String, (String, Expr)]) extends Expr}

We will use labeled union types or just union types to accomplish this. A union type is a map from strings to types, where the strings are the names of constructors (like case classes), mapped to what type their argument is supposed to be. We will give abstract syntax for

For example,

\texttt{TUnion(Map("CInt" -> TInt, "CBool" -> TBoolean))}

The power of using union types is flexibility. For example, in normal \texttt{SCALA} we can write a function which takes in an \texttt{AEExpr} as an argument without knowing which subtypes of \texttt{AEExpr} it will actually be called on. The \texttt{Inj} expression (inject expression) is a mechanism for “upcasting” an expression to a union type, while storing its constructor name so that it can be matched against the correct constructor pattern later.

Using our typing rules:

\texttt{B(true): TBoolean}

\texttt{Inj(TUnion(Map("CInt" -> TInt, "CBoolean" -> TBoolean)), "CBoolean", B(true)):
TUnion(Map("CInt" -> TInt, "CBoolean" -> TBoolean))}

As a brief example, suppose we wanted to write a function which can take in either Ints or Booleans, and prints their numerical representation.
Fun("z", TUnion(Map("CInt" -> TInt, "CBoolean" -> TBoolean)),
    Match(TUnit, "z",
        Map("CInt" -> ("x", Print(Var("x"))),
            "CBoolean" -> ("x", If(Var("x"), Print(N(1)), Print(N(0)))))))

The abstract syntax gets large, so we will show an example of reduction steps in applying
this function using the more terse syntax in the reduction rules. The following one-step
evaluation judgments hold in our system.

Step 1:
\[
((z:\texttt{Unit}, CInt, CBool : \texttt{Boolean}]) \Rightarrow \\
\texttt{matchUnit} \\
\texttt{case} CInt:x \Rightarrow \texttt{print}(x) \\
\texttt{case} CBool:x \Rightarrow \text{if} (x) \texttt{print}(1) \text{ else } \texttt{print}(0) \texttt{) (inj[...](CBool:true))}
\]

\[
\rightarrow \\
\texttt{inj[...](CBool:true) matchUnit} \\
\texttt{case} CInt:x \Rightarrow \texttt{print}(x) \\
\texttt{case} CBool:x \Rightarrow \text{if} (x) \texttt{print}(1) \text{ else } \texttt{print}(0)
\]

Step 2:
\[
\texttt{inj[...](CBool:true) matchUnit} \\
\texttt{case} CInt:x \Rightarrow \texttt{print}(x) \\
\texttt{case} CBool:x \Rightarrow \text{if} (x) \texttt{print}(1) \text{ else } \texttt{print}(0)
\]

\[
\rightarrow \\
\text{if} (\texttt{true}) \texttt{print}(1) \text{ else } \texttt{print}(0)
\]

Step 3:
\[
\texttt{if} (\texttt{true}) \texttt{print}(1) \text{ else } \texttt{print}(0) \rightarrow \texttt{print}(1)
\]

Step 4:
\[
\texttt{print}(1) \rightarrow ()
\]