Summary of Moore on Huet-Lang

Nathan Carter

28 March 2019

1 Expressions and replacement

Assume we have a format for storing mathematical (or other kinds of formal) expressions as trees. OpenMath is one such format. Call the set of things that format can express (unsurprisingly) *expressions* and write it as E.

In an expression, one can replace all free occurrences of a variable with another expression. We write this as $e[a := b]$, pronounced "e with a replaced by b." Specifics of this definition can be provided if needed.

We assume the existence of an algorithm for checking if two given expressions are α-equivalent. One example algorithm is to convert both to a canonical form by replacing bound variables with variables from the list $v_1, v_2, v_3, \ldots$ in the order encountered in a depth-first traversal, then comparing the expressions for actual equality.

We assume the existence of an algorithm for performing β-reduction on expressions of the form $(\lambda x.B)(a)$ by computing $B[x := a]$. Unless a contains x free, in which case we must first α-convert $\lambda x.B$ to some $\lambda x'.B[x := x']$ for an entirely new variable x' that does not occur in B or a before doing the β-reduction.

2 Metavariables and functions

Assume we can mark some variables as those that will be used for substitution, so-called *metavariables,* by attaching attributes to particular leaves in an expression tree, so that it is obvious, when examining an expression, which things in it are metavariables. Further assume that we will not attach this attribute to anything but a free variable. Call expressions containing metavariables *patterns.*

Assume we can mark some metavariables as those that will stand for functions from E to E, and we will use the usual notation $f(x)$, when $f : E \to E$ and $x \in E$, to represent the expression resulting from applying the function.

For instance, the function $\lambda x.x + y$ can be considered a function from E to E by treating its body as if it were an element of E into which we will substitute x once it is provided. If we evaluate such a function on, say, the number 3, we get the expression $3+y$. To be specific, $\lambda x.x+y$ is to be interpreted as the function that, given any input a, performs $(x+y)[x := a]$, yielding the unevaluated expression $a + y$.

This is unfortunately ambiguous, because the expression language itself contains function application as well, which is entirely different. In the expression language, the notation $f(x)$ is itself an expression, of type "function application." But in the metalanguage, the same notation indicates the computation of an expression by evaluating a function that maps into E, as defined in the previous paragraphs.

So when reading $f(x)$, it is important to know whether $f \in E$ or $f : E \to E$, to disambiguate. If $f \in E$, then $f(x)$ is a function-application expression in E with two children, f and x. If $f : E \to E$, then $f(x)$ could be anything; it is still a member of E, but it is whatever member of E f yields when applied to x. And in such a case, f itself is not a member of E, though x is.

3 Instantiating metavariables

An *instantiation* of metavariables is a mapping from the set of metavariable names to the set of expressions; we typically consider just those instantiations that can be expressed as finite sets of ordered pairs, because we will be writing algorithms to construct such instantiations.

To be specific, the recursive process $Apply(I, p)$ of applying an instantiation I to a pattern p is as follows.

1. If p is a metavariable in the domain of I then return $I(p)$.
2. If p is a non-metavariable, or is a metavariable not in the domain of I, or is a constant/symbol, then return p itself (or typically a copy of p).
3. If p is of the form $f(a_1, \ldots, a_n)$ then return the β reduction of

$$Apply(I, f)(Apply(I, a_1), \ldots, Apply(I, a_n)).$$

   Here, β reduction has the usual meaning: any term of the form $(\lambda x.B)(a)$ is replaced by $B[x := a]$.

4 Matching

Matching is the process of taking a pattern p and an expression e and creating one or more instantiations I such that $Apply(I, p) = e$. More generally, we take a set of pairs $\{(p_1, e_1), \ldots, (p_n, e_n)\}$ and aim to compute "all" instantiations I such that $\forall i \in \{1, \ldots, n\}, Apply(I, p_i) = e_i$. Thus the result of a matching algorithm is a set of instantiations, and the set may be empty. We call a set of pattern-expression pairs, like $\{(p_1, e_1), \ldots, (p_n, e_n)\}$, a *matching challenge.*

Given a matching challenge, we now define the operation of making a substitution into its patterns. (It does not make sense to make a substitution into its expressions, since they contain no metavariables.) Thus for a matching challenge C of the form given above, we define $C[a := b]$ to be the set

$$\{(p_1[a := b], e_1), \ldots, (p_n[a := b], e_n)\},$$

plus an annotation of the fact that we have done the substitution $a := b$. That is, the result is not just a set, but a set "with a note attached."

We expect that we may do several substitutions during the course of solving a matching challenge, and would like to be able to recover later the list of substitutions we did en route to the solution. In fact, for a matching challenge C, let's write $sub(C)$ to mean the list of substitutions that has been done during the life of C, each of which has been noted in C.

We write $\pi_{k,i}$ for the i^{th} projection function on k arguments, $\pi_{k,i} = \lambda x_1, \ldots, x_k.x_i$.

The Huet-Lang algorithm (as summarized in [1]) proceeds as follows. Given a matching challenge $C = \{(p_1, e_1), \ldots, (p_n, e_n)\}$, with each p_i a pattern and each e_i an expression, we compute $Matches(C)$ as follows. (The result will be a list, whose interpretation is explained further below. I use square brackets [...] to denote lists.)

1. **Remove already-matching pairs:**

   If some (p_i, e_i) satisfies $p_i = e_i$ (and thus p_i contains no metavariables) then return $Matches(C - \{(p_i, e_i)\})$. (Here, the binary minus, $-$, represents set difference, and two expressions are considered equal if they are α-equivalent.)

2. **Assign metavariables if required:**

   If some p_i is a metavariable then return

   $$Matches\Big((C - \{(p_i, e_i)\})[p_i := e_i]\Big).$$

3. **Recur into expression trees:**

   If some (p_i, e_i) is of the form $(f(p'_1, \ldots, p'_k), f(e'_1, \ldots, e'_k))$ with f not a metavariable (and of course each p'_i a pattern and each e'_i an expression), then return

   $$Matches\Big(C - \{(p_i, e_i)\} \cup \{(p'_1, e'_1), \ldots, (p'_k, e'_k)\}\Big).$$

   As a special case of this, if f is a quantifier, then perform any needed α conversion on p_i to ensure its bound variables match those of e_i before proceeding, because we aim to treat α-equivalent expressions as equal.

4. **Consider an unassigned function metavariable, which may be (a) just a constant function, (b) just a projection function, or (c) a more complex function:**

   If some p_i is of the form $f(p'_1, \ldots, p'_k)$ (with each p'_i a pattern), but the previous case does not hold, then proceed as follows:

   (a) Compute the temporary result

   $$L_1 := Matches(C[f := \lambda v_1.e_i]),$$

   which will be a list. (This handles the case where f might be just a constant function. The v_1 should be a new variable.)

(b) Compute the temporary result L_2, the list obtained by concatenating all the lists $Matches(C[f := \pi_{k,i}])$ for $1 \leq i \leq k$. (This handles the case where f might be just a projection function.)

(c) If e_i is of the form $g(e'_1, \ldots, e'_m)$ then let

$$E = \lambda v_1, \ldots, v_k.g(h_1(v_1, \ldots, v_k), \ldots, h_m(v_1, \ldots, v_k))$$

and let $L_3 := Matches(C[f := E])$. (This handles the case where f might be a more complex function. Each v_i should be a new variable.)

(d) Return $L_1 * L_2 * L_3$ as the result, where $*$ means list concatenation.

5. **Success case:**

   If C is empty, return $[sub(C)]$, that is, the list containing just one entry, and that one entry is the list of substitutions made during the life of C, which is the one solution we have found to the matching problem.

6. **Failure case:**

   If none of these cases apply, return [ ], that is, the empty list, to show that we have found precisely zero solutions.

When this process is done, it will yield a list, each entry of which (if there are any) will be a list of substitutions that instantiate all the metavariables in the original matching problem in a way that solves the problem. If there are any entries in the list, then the original matching challenge is solvable. There may be more than one entry in the list, which will mean more than one solution.

Note that one can sort the set C to prioritize items from Case 4 lowest, thus creating many recursive branches only if necessary. (Since many branches of the recursive exploration fail, try everything before branching, since the branch you're on may fail, saving you the trouble of more branches, i.e., pruning the tree sooner.) Furthermore, prioritizing highly any pair (p_i, e_i) in which p_i contains no metavariables is important, because if $p_i \neq e_i$ in such a case, you can skip all cases except Case 6.

5 Example Runs

This section shows a few example runs of the *Matches* algorithm. I use capital letters for metavariables and lower-case letters for regular variables. I use infix notation for common operators (e.g., $x + y$, $p \wedge q$) but its functional representation would be more like $+(x, y)$ and $\wedge(p, q)$.

Example 1

$$
\begin{aligned}
& Matches(\{(P \wedge Q, a \wedge (b \vee c))\}) & \\
= {} & Matches(\{(P, a), (Q, b \vee c)\}) & 3 \\
= {} & Matches(\{(Q, b \vee c)\}) & 2,\ P := a \\
= {} & Matches(\{\}) & 2,\ Q := b \vee c \\
= {} & [\{\}] & 5
\end{aligned}
$$

Result: One match, in which is recorded $P := a, Q := b \vee c$. (The matching challenge written $\{\}$ in the last line does not include these annotations, just for brevity, but they were noted in the right-hand column of the computation.)

Example 2

$$
\begin{aligned}
& Matches(\{((X + Y) \cdot (X - Y), (3 + k) \cdot (3 - p))\}) & \\
= {} & Matches(\{(X + y, 3 + k), (X - Y, 3 - p)\}) & 3 \\
= {} & Matches(\{(X, 3), (Y, k), (X - Y, 3 - p)\}) & 3 \\
= {} & Matches(\{(Y, k), (3 - Y, 3 - p)\}) & 2,\ X := 3 \\
= {} & Matches(\{(3 - k, 3 - p)\}) & 2,\ Y := k \\
= {} & Matches(\{(3, 3), (k, p)\}) & 3 \\
= {} & Matches(\{(k, p)\}) & 1 \\
= {} & [\{(k, p)\}] &
\end{aligned}
$$

Result: No match, because the only matching challenge in the list is one that is nonempty.

Example 3

$$
\begin{aligned}
& Matches(\{(P(1) \wedge P(2), 0 \neq 1 \wedge 0 \neq 2)\}) & \\
= {} & Matches(\{(P(1), 0 \neq 1), (P(2), 0 \neq 2)\}) & 3
\end{aligned}
$$

Now we must consider the three parts of Case 4 for the pair $(P(1), 0 \neq 1)$.

$$\begin{array}{ll}
L_1 = Matches(\{(0 \neq 1, 0 \neq 1), (0 \neq 1, 0 \neq 2)\}) & 4(a),\ P := \lambda V_1.0 \\
\quad = Matches(\{(0 \neq 1, 0 \neq 2)\}) & 1 \\
\quad = [\,] & 6 \\
L_2 = Matches(\{(1, 0 \neq 1), (2, 0 \neq 2)\}) & 4(b),\ P := \lambda V_1.V_1 \\
\quad = [\,] & 6 \\
L_3 = Matches(\{(H_1(1) \neq H_2(1), 0 \neq 1), & \\
\qquad\qquad (H_1(2) \neq H_2(2), 0 \neq 2)\}) & 4(c),\ P := \lambda V_1.H_1(V_1) \neq H_2(V_1) \\
\quad = Matches(\{(H_1(1), 0), (H_2(1), 1), & \\
\qquad\qquad (H_1(2) \neq H_2(2), 0 \neq 2)\}) & 3 \\
\quad = Matches(\{(H_1(1), 0), (H_2(1), 1), & \\
\qquad\qquad (H_1(2), 0), (H_2(2), 2)\}) & 3
\end{array}$$

While L_1, L_2 yielded no solutions, L_3 breaks into parts by Case 4 for the pair $(H_1(1), 0)$. But only parts (a) and (b) of that case apply.

$$\begin{array}{ll}
L_1 = Matches(\{(0, 0), (H_2(1), 1), (0, 0), (H_2(2), 2)\}) & 4(a),\ H_1 := \lambda V_1.0 \\
\quad = Matches(\{(H_2(1), 1), (0, 0), (H_2(2), 2)\}) & 1 \\
L_2 = Matches(\{(1, 0), (H_2(1), 1), (2, 0), (H_2(2), 2)\}) & 4(b),\ H_1 := \lambda V_1.V_1 \\
\quad = [\,] & 6
\end{array}$$

While L_2 yielded no solutions, L_1 breaks into parts by Case 4 for the pair $(H_2(1), 1)$. But only parts (a) and (b) of that case apply.

$$\begin{array}{ll}
L_1 = Matches(\{(1, 1), (0, 0), (1, 2)\}) & 4(a),\ H_2 := \lambda V_1.1 \\
\quad = Matches(\{(1, 2)\}) & 1 \text{ twice} \\
\quad = [\,] & 6 \\
L_2 = Matches(\{(1, 1), (0, 0), (2, 2)\}) & 4(b),\ H_2 := \lambda V_1.V_1 \\
\quad = Matches(\{\}) & 1 \text{ thrice} \\
\quad = [\{\}] & 5
\end{array}$$

Thus the only solution found was the one down the path $P := \lambda V_1.H_1(V_1) \neq H_2(V_1), H_1 := \lambda V_1.0, H_2 := \lambda V_1.V_1$. If we substitute the temporary variables into the body of P, we find the solution is $P := \lambda V_1.0 \neq V_1$.

6 Additional requirements

Here are matching problems that this implementation will need to be able to solve, together with some comments. Note that most of these are complex enough that I have not tried to trace the execution of *Matches* on them by hand. I use the constant w as a generic wrapper object.

Pattern	**Expression**	**Result**
$w(\forall x.P, P(T))$	$w(\forall r.r^2+1>0,$ $(-9)^2+1>0)$	$P := \lambda V_1.V_1^2+1>0$
$w(w(X, P(X)),$ $\forall x.P(x))$	$w(w(k, k+1<5),$ $\forall s.s+1<5)$	$X := k$ $P := \lambda V_1.V_1+1<5$
$w(\exists x.P(x),$ $\forall y.(P(y)\rightarrow Q),$ $Q)$	$w(\exists x.x^3=-1,$ $\forall x.x^3=-1\rightarrow x<5,$ $x<5)$	no match, because the x in the second line becomes bound

7 Desired API

Ideally, we would like the contents of this document implemented as follows.

1. Permit the marking of variables as metavariables (and the detection of such a status) using OpenMath attributes.

2. Create a class that will embody a matching challenge, and let us create new empty instances of it. It should provide a `clone()` method for making exact copies of itself (including any solution annotations that have been computed.)

3. That class should permit the addition of pattern-expression pairs as new constraints, with a function named something like `addConstraint()`.

4. That class should permit asking whether it has solutions, with functions named something like `isSolvable()`, `numSolutions()`, and `getSolutions()`.

5. It should not assume that all calls to `addConstraint()` will take place before calls to `isSolvable()`, `numSolutions()`, and `getSolutions()`. Thus if a constraint is added, those results may need to be recomputed the next time they are queried, and the constraint itself may need to be processed with any substitutions already computed.

6. It should also provide an iterator-style interface that uses `yield` to produce solutions, rather than computing them all up front.

References

[1] J. Strother Moore, *Automatically Computing Functional Instantiations*. Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications, pp.11–19. 2009. doi:10.1145/1637837.1637839