# A Limited First-Order Matching Algorithm

Nathan Carter

September 3, 2018

**Abstract**

This document describes a limited first-order matching algorithm. This algorithm is implemented in JavaScript in an `npm` package and this document serves as documentation for the theory behind the implementation.

## 1 Syntax and language

**Definition 1** (expression)**.** *An* expression *will be any of the ordinary types of mathematical expression in any language sufficiently robust enough to represent operations and quantifiers, such as the language of first-order logic.*

If we need to be precise, we can consider expressions to be OpenMath[1] expressions, with some finite set of common mathematical symbols, and no attributes attached. We will speak of expressions as having syntax trees and having subexpressions, with the usual meaning of those terms, not defined precisely here. Again, if precision is required, just assume we're using OpenMath.

Let $E$ denote the set of all expressions.

**Definition 2** (address)**.** *An* address *of a subexpression $x$ in an expression $y$ is the sequence of steps one would take from the root of $y$'s syntax tree to reach $x$.*

If the immediate subexpressions of any given expression are ordered from left-to-right in the order written, then each "step" can simply be an integer index into that ordered set of immediate subexpressions, making an address a vector of integers.

In such a situation, we might consider the expression $3x + t^2$ to have a syntax tree that we might write in LISP-like notation as `(plus (times 3 x) (power t 2))` and thus the $x$ would have address $\langle 1, 2\rangle$ within the larger expression, using zero-based indices. The details are unimportant; what matters is that every subexpression has an address and such addresses can be used to look up subexpressions. The empty vector $\langle\rangle$ indicates the address of an expression within itself.

Given an expression $e$ and an address $a$ into it, we write $e[a]$ to mean the subexpression at address $a$ within $e$. Note that the syntax $e[a]$ is *not* itself a syntactic form within $E$, but rather a notation used in this document to describe subexpressions of $e$, each of which is of course in $E$. Thus while the syntax $e[a]$ is not in $E$, the result of evaluating that syntax is in $E$.

**Definition 3** (uniform)**.** *An address set $A$ is* uniform on *an expression $e$ iff $\forall a, a' \in A, e[a] = e[a']$.*

If you consider the LISP-like notation just introduced, you may think that expressions are defined solely by their hierarchy of children. This is one model of expressions, but we permit something broader.

**Definition 4** (expression type)**.** *Each expression can be marked with a* type, *one of a small finite list of values.*

For example, OpenMath has multiple types of compound expressions, including one for function or operator application, another for quantifier binding, another for error objects, and a few rarely used others. We assume this format herein, but if you do not wish to use it, just assume any set of types, as long as it at least distinguishes binding expressions from non-binding ones. We also assume herein that each binding expression binds one variable, but it is not difficult to generalize this if needed.

Expression type is the only additional piece of information that we will take into account in this document. Consequently, we have the following statement about identity of expressions.

**Definition 5** (expression equality)**.** *Two expressions are* equal *iff their types are equal, their children arrays have the same length (which can be zero if the expressions are atomic), and all their corresponding children are pairwise equal (which is vacuously true if the expressions are atomic).*

**Definition 6** (difference)**.** *Define the* difference *between two expressions $e_1$ and $e_2$ to be the set $A$ of addresses with the following properties.*

1. *For each $a \in A$, $e_1[a] \neq e_2[a]$.*
2. *The elements of $A$ are of maximal length, in the sense that the difference between $e_1[a]$ and $e_2[a]$ can be observed without descending into children of $e_1[a]$ and $e_2[a]$. (That is, either the type or number of children of $e_1[a]$ and $e_2[a]$ differ, or they are distinct atomic expressions.)*

For instance, if we compute the difference between $x+y$ and $x+3$, we would not have the address $\langle\rangle$ in the result, because although $(x+y)[\langle\rangle] \neq (x+3)[\langle\rangle]$, it requires inspecting the third child of each of $x+y$ and $x+3$ to see the difference. Instead, the difference is $\{\langle 2\rangle\}$. This is what is meant by the elements having maximal length.

Note that this requirement occasionally makes the difference larger. Consider the difference of 6(1+2) with 6(3+4) (whose syntax trees are like `(times 6 (plus 1 2))` and `(times 6 (plus 3 4))`). The set $\{\langle 2\rangle\}$ satisfies the first half of Definition 6 but does not satisfy the second half, the maximal length requirement. The difference must instead be $\{\langle 2,1\rangle, \langle 2,2\rangle\}$.

**Definition 7** (parent)**.** *Every non-empty address $a$ can be converted into a* parent *address by simply leaving off its last entry. Thus the parent of $\langle a_1,\ldots,a_n\rangle$ is $\langle a_1,\ldots,a_{n-1}\rangle$ and the parent of $\langle a_1\rangle$ is $\langle\rangle$.*

We also speak of ancestor and descendant addresses, with the obvious meanings taken to be paths of parent/child relationships. We extend those ideas with the following two definitions as well.

**Definition 8** (ancestor address set)**.** *Given two sets $A$ and $A'$ containing only addresses (as in Definition 2), we say that $A$ is an* ancestor *of $A'$ if every address in $A'$ has an ancestor in $A$ and every address in $A$ has a descendant in $A'$.*

For example, $\{\langle 1\rangle, \langle 2,0\rangle\}$ is an ancestor of $\{\langle 1,1,4\rangle, \langle 2,0\rangle\}$, but not of $\{\langle 1\rangle, \langle 2,2\rangle\}$. Furthermore, $\{\langle\rangle\}$ is an ancestor of every nonempty address set.

**Definition 9** (metavariable)**.** *Introduce a new set of variables, not in $E$, called* metavariables*. One way to do so is to take the variables in $E$ and add to each a flag denoting "metavariable," resulting in a new syntactic object not in $E$.*

In OpenMath, we can attach an attribute whose key is "metavariable" and whose value is "true" to mark variables as metavariables. The exact way of marking variables is unimportant, but we will assume that both reading and writing this attribute is a trivial operation. Call the set of metavariables $M$, a set which is disjoint from $E$.

**Definition 10** (expression function, expression function application)**.** *When $P : E \to E$ we call $P$ an* expression function. *When $x \in E$ and $P : E \to E$, we may write $P(x)$ with the usual meaning, the application of $P$ to $x$. However, when $P \in M$, we may also write $P(x)$, and we call such a form an* expression function application. *Expression function applications cannot contain other expression function applications (no nesting, as in $P(Q(x))$).*

Because expression function application is not the same operation as function application in $E$, expression function applications are not themselves elements of $E$. So the syntactic form $P(a)$ is not in $E$, for two reasons: first, because the application operation is not in $E$, and second, because $P \in M$ and $M \cap E = \emptyset$.

**Definition 11** (metaexpressions)**.** *Metavariables and expression function applications are called* metaexpressions.

Let $E'$ be the set of expressions in $E$, plus any copies of those expressions with subexpressions replaced by metaexpressions. For instance, if $x + y$ is in $E$, then $P(3a) + y$ is in $E'$, and so is $X + y$, where $X \in M$. We will not write $f(x)$ to mean function application in $E$, in order to prevent ambiguity with respect to expression function application. But $E$ permits function application, so in the rare cases that we need to mention it, I will write $(f\ x)$ for function application in $E$, borrowing from LISP-like languages. (More generally, we could write $(f\ a_1\ \ldots\ a_n)$, but this document almost never needs to mention function application in $E$.) Since both $f$ and $x$ are subexpressions of $(f\ x)$, if we had $P, Q \in M$, we would have $(P\ Q) \in E'$.

All metavariables and all expression function applications are in $E'$, but $E'$ is more than just $E$ plus metaexpressions; it also permits metaexpressions inside otherwise-normal expressions, as in the examples just given. Furthermore, expression function applications always contain at least one metavariable, but not the reverse (obviously), and expression function applications may not contain other expression function applications, as already stated in Definition 10.

## 2 Problem statement

**Definition 12** (constraint)**.** *A* constraint *$c$ is a pair $(c_1, c_2) \in E' \times E$. It expresses the idea that we will want the expression $c_2$ to fit the "pattern" given by $c_1$.*

We have not yet defined what we mean by "pattern," but Definition 16 will make it precise, by viewing metavariables in $c_1$ as parameters that can be filled by elements of $E$. Naturally, when we speak of a *constraint set $C$*, we mean a set of constraints, or equivalently, a subset of $E' \times E$.

**Definition 13** (type consistent)**.** *A constraint set $C$ is* type inconsistent *if some metavariable $V \in M$ occurs in $C$ at least twice,*

1. *once as the head of an expression function application (meaning that $V$ must be instantiated with an expression function), and*
2. *elsewhere* not *as the head of an expression function application (meaning that $V$ must be instantiated with an element of $E$).*

*A constraint set $C$ is* type consistent *otherwise.*

**Definition 14** (potential solution)**.** *A* potential solution *is a partial function $s : M \to E$; it can therefore be used to instantiate metavariables.*

For convenience, we will write $s(e)$ for any $e \in E'$ to mean the expression resulting from the simultaneous replacement, in $e$, of all metavariables $m$ appearing in both $e$ and the domain of $s$ with the value of $s(m)$.

**Definition 15** (extending a solution)**.** *We say that a potential solution $s$* extends *a potential solution $s'$ if, when viewed as sets of ordered pairs, $s \supseteq s'$. We can also say* properly extends *for $s \supsetneq s'$.*

**Definition 16** (complete solution)**.** *A* complete solution *to a constraint set $C$ is a potential solution $s$ such that for any $s'$ extending $s$, every $(c_1, c_2) \in C$ satisfies $s'(c_1) = c_2$.*

Notice that this definition means that a constraint set $C$ is taken to be the conjunction of its individual elements.

Because we will recursively define a way to compute solutions, we will construct complete solutions by building up potential solutions until they are complete. Consequently, we will speak of solutions not only to constraint sets $C$, but to pairs $(C, s)$ in which $C$ is a constraint set and $s$ is a potential solution. In that case, what we mean by a solution is a solution to $C$ that extends $s$. Alternately, one can view it as a solution to $C \cup s$, when viewing $s$ as a set of ordred pairs, a subset of $M \times E$, which in turn is a subset of $E' \times E$, just as $C$ is.

**Definition 17** (full solution set)**.** *The* full solution set *for a constraint set $C$ is the set of all minimal complete solutions for $C$. A solution is minimal if none of its proper subsets is a complete solution. We will use the notation $F(C)$ for the full solution set for $C$.*

Similarly, a full solution set for the pair $(C, s)$ is the set of all minimal complete solutions for the pair $(C, s)$, that is, for the constraint set $C \cup s$. We write it as $F(C, s)$.

Note that it is trivial to take a set $S$ containing all the complete solutions to a pair $(C, s)$ and convert it into the full solution set, by simply removing those solutions that are not minimal. We can tell whether a solution is minimal by checking to see if any of its proper subsets is an element of $S$. We can write this as an operation.

$$\min S = \{s \in S \mid \forall s' \in S,\ \text{if } s' \subseteq s \text{ then } s' = s\}$$

We will occasionally want to form unions of minimal sets, for which the following lemma is handy.

**Lemma 1.** *Let $C_0, \ldots, C_n$ be constraint sets and $s_0, \ldots, s_n$ be potential solutions. Consider $(C_0, s_0)$ separately from the rest of the list as follows: Assume that $s$ is a complete solution for $(C_0, s_0)$ iff for some $i \in \{1, \ldots, n\}$, $s$ is a complete solution for $(C_i, s_i)$. Define the shorthand*

$$U = \bigcup_{1 \leq i \leq n} F(C_i, s_i).$$

*Then $F(C_0, s_0) = \min U$.*

*Proof.* First assume $s \in F(C_0, s_0)$. Then $s$ is a complete solution for $(C_0, s_0)$ and so we have some $i \in \{1, \ldots, n\}$ such that $s$ is a complete solution for $(C_i, s_i)$. Furthermore, for any $s' \in F(C_0, s_0)$, if $s' \subseteq s$ then $s' = s$. Now suppose towards a contradiction that for some $j \in \{1, \ldots, n\}$ and some $s' \in F(C_j, s_j)$, $s' \subsetneq s$. Then $s'$, as a complete solution of $(C_j, s_j)$, would be a complete solution of $(C_0, s_0)$ by the assumption of the lemma, which would prevent $s$ from being in $F(C_0, s_0)$ by minimality. This contradiction means that for any $j \in \{1, \ldots, n\}$ and any $s' \in F(C_j, s_j)$, if $s' \subseteq s$ then $s' = s$. We can also write this as $\forall s' \in U$, if $s' \subseteq s$ then $s' = s$. This proves that $s \in \min U$, completing the left-to-right direction of the inclusion.

Now assume that $s \in \min U$. Then $s \in U$ and $\forall s' \in U$, if $s' \subseteq s$ then $s' = s$. From the first of these, we have that there is some $i \in \{1, \ldots, n\}$ such that $s \in F(C_i, s_i)$, and so $s$ is complete for $(C_i, s_i)$. By the assumption of the lemma, $s$ is therefore complete for $(C_0, s_0)$. Now assume toward a contradiction that some $s' \subsetneq s$ is also complete for $(C_0, s_0)$. Then for some $j \in \{1, \ldots, n\}$, $s'$ is complete for $(C_j, s_j)$, and some subset $s'' \subseteq s'$ is an element of $F(C_j, s_j)$. Thus $s'' \in U$, but because $s'' \subsetneq s$, this prevents $s$ from being in $\min U$, a contradiction. Thus for any complete solution $s'$ for $(C_0, s_0)$, if $s' \subseteq s$ then $s' = s$. Thus $s \in F(C_0, s_0)$, completing the right-to-left direction, and thus the proof. □

**Definition 18** (respecting free variables)**.** *Given a potential solution $s$ and a pattern $p \in E'$, we say that $s$* respects free variables in $p$ *to mean that*

1. *for any two addresses $a, a'$ into $p$, with $a$ an ancestor of $a'$, if $p[a']$ is free in $p[a]$, then $s(p)[a']$ is free in $s(p)[a]$, and*

2. *for any expression function application $P(x)$ appearing in $P$, with $s(P) = \lambda v.b$, $s(x)$ is free to be substituted $b[v = s(x)]$.*

We will want solutions that respect free variables, but we will ignore that for now, returning to it in Section 7.

# 3 Specification

I claim that the following recursive equations respect the definition of $F$ and give us an outline for how to implement the computation of $F$. I prove the claim as Theorem 2, below. In these recursive equations, $C$ is any constraint set and $s$ is any potential solution. The cases are to be read in order, so that case $n$ is considered only if cases 1 through $n-1$ do not apply.

Technically, the function $F$ defined here is ambiguous, because in some cases it chooses an arbitrary constraint (or two) from the set $C$ to process next, an operation that is not deterministic. We ignore this small problem, because one can fix it by treating $C$ as a list instead and always choosing the earliest entry to remove ambiguity. This requires replacing $\cup$ with list concatenation (removing duplicate entries) as well. We prefer the simpler set-theoretic notation, but the reader is free to think in terms of lists instead.

1. **Base case:**
$$F(\{\}, s) = \{s\}.$$

2. **Atomic case:** Given any $a, e \in E$ with $a$ atomic,
$$F(C \cup \{(a, e)\}, s) = \begin{cases} F(C, s) & \text{if } a = e \\ \\ \{\} & \text{otherwise.} \end{cases}$$

3. **Non-atomic case:** Given any $e \in E$ and $t \in E'$ with $t$ non-atomic, not an expression function application, and having children $t_1, \ldots, t_n$,
$$F(C \cup \{(t, e)\}, s) = \begin{cases} F(C \cup \{(t_1, e_1), \ldots, (t_n, e_n)\}, s) & \text{if } t, e \text{ have the same type and} \\ & \quad e \text{ has } n \text{ children, } e_1, \ldots, e_n \\ \{\} & \text{otherwise.} \end{cases}$$

   (Expression types are as in Definition 4.)

4. **Inconsistent case:** If any $m_1, m_2 \in M$, $t_1, t_2 \in E'$, $e_1, e_2 \in E$, then
$$F(C \cup \{(m_1(t_1), e_1), (m_1, e_2)\}, s) = \{\}.$$

   Furthermore, if $m_1$ appears in $t_1$, then
$$F(C \cup \{(m_1(t_2), e_1), (m_2(t_1), e_2)\}, s) = \{\}$$

   and
$$F(C \cup \{(m_1(t_1), e_1)\}, s) = \{\}.$$

   (When $t_1$ appears inside an expression function application, we know by Definition 10 that it cannot contain any expression function applications itself. Thus it must require $m_1$ to match an expression, which may create a type inconsistency. This case is discussed in detail in Lemma 2. See also Definition 13.)

5. **Metavariable case:** Given any $m \in M$ and $e \in E$,

$$F(C \cup \{(m,e)\}, s) = \begin{cases} \{\} & \text{if } m \text{ is in } s \text{ and } s(m) \neq e \\ \\ F(C, s \cup \{(m,e)\}) & \text{otherwise.} \end{cases}$$

6. **First of two expression function application cases:** Given any $m \in M$, $e_1 \neq e_2 \in E$, and $t_1, t_2 \in E'$, we will now handle cases of the form $F(C \cup \{(m(t_1), e_1), (m(t_2), e_2)\}, s)$. Compute the difference between $e_1$ and $e_2$ (as in Definition 6) and call it $D$.

   Let $A$ be the set of all ancestor sets to $D$ that are uniform on both $e_1$ and $e_2$. Let $v$ be a new variable not appearing in $C$ or $s$. Given any set $S$ of addresses, write $e_1[S/v]$ to mean the simultaneous replacement of each $e_1[a]$ with $v$ for all addresses $a \in S$.

   Compute $A'$ to be the subset of $A$ containing just those $S \in A$ such that $s \cup \{(m, \lambda v.e_1[S/v])\}$ is a partial function. That is, either $m$ is not in $s$ yet (and thus we can add it) or $s(m) = \lambda v.e_1[S/v]$ (so the union is simply $s$ again). (In the equation $s(m) = \lambda v.e_1[S/v]$, both sides of the equations are functions, not syntactic structures, so $\alpha$-equivalence is implied in the comparison.) Let the union $s \cup \{(m, \lambda v.e_1[S/v])\}$ be called $s_S$.

   For each $S \in A'$, fix a chosen element $a_S \in S$. (Note that no $S \in A$ is empty, because $e_1 \neq e_2$, so the set $D$ is nonempty, and thus all its ancestor sets are also nonempty.) Then

$$F(C \cup \{(m(t_1), e_1), (m(t_2), e_2)\}, s) = \min\left(\bigcup_{S \in A'} F(C \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\}, s_S)\right).$$

   Note that the choice of $a_S \in S$ is immaterial, because $S$ is uniform on $e_1$ and $e_2$, so $e_1[a_S]$ and $e_2[a_S]$ are independent of the choice of $a_S$. Also note that $A'$ cannot be empty, because $\{\langle\rangle\}$ is an ancestor of any difference set $D$, and $\{\langle\rangle\}$ is uniform on every expression, thus passing the test for membership in $A'$.

7. **Second of two expression function application cases:** Given any $m \in M$, $e \in E$, and $t_1, t_2 \in E'$, we will now handle cases of the form $F(C \cup \{(m(t_1), e), (m(t_2), e)\}, s)$. Let $v$ be a new variable not appearing in $C$ or $s$. Given any set $S$ of addresses, write $e[S/v]$ to mean the same as in the previous case.

   Let $U$ be the set of subexpressions of $e$, and for each $u \in U$, write $A_u$ to mean the set of addresses into $e$ at which $u$ appears. Let $s_S$ be defined as in the previous case, but with $e$ instead of $e_1$. Then

$$F(C \cup \{(m(t_1), e), (m(t_2), e)\}, s) =$$
$$\min\left(\left(\bigcup_{u \in U, S \subseteq A_u, S \neq \emptyset} F(C \cup \{(t_1, u), (t_2, u)\}, s_S)\right) \cup F(C, s \cup \{(m, \lambda v.e)\})\right),$$

   where the indexed union is limited to cases for which $s_S$ is a partial function, and the second half of the non-indexed union is omitted if $s \cup \{(m, \lambda v.e)\}$ is not a partial function.

8. **Only remaining case:** The only case that remains is when the constraint set is nonempty, each of its constraints has an expression function application as the left-hand side, and yet no expression function appears in more than one constraint. We therefore single out one expression function application and handle it, and recursion will handle them all one at a time.

   If $m$ is in $s$ then

$$F(C \cup \{(m(t), e)\}, s) = F(C \cup \{(s(m)(t), e)\}, s),$$

and otherwise

$$F(C \cup \{(m(t), e)\}, s) = \min\left(\left(\bigcup_{u\in U, S\subseteq A_u, S\neq\emptyset} F(C \cup \{(t, u)\}, s_S)\right) \cup F(C, s \cup \{(m, \lambda v.e)\})\right),$$

where $U$, $A_u$, and $s_S$ are as in the previous case, the indexed union is limited to cases for which $s_S$ is a partial function, and the second half of the non-indexed union is omitted if $s \cup \{(m, \lambda v.e)\}$ is not a partial function.

# 4 Results

**Theorem 1.** *The recursive definition of $F$ above, evaluated on any finite $C$, will terminate.*

*Proof.* Every clause references a "simpler" use of the function $F$, meaning that either the constraint set $C$ has shrunk in cardinality or some element of $C$ has been replaced by strictly smaller elements of $C$. If we imagine $c(C)$ to be the cardinality of $C$ and $d(C)$ to be the average tree depth of all left-hand-side expressions in $C$, and $e(C)$ to be the number of constraints in $C$ that have an expression function application as the left hand side, then we can dictionary order the set of all possible $C$ sets with $c(C)$ as the primary component, $e(C)$ as the secondary, and $d(C)$ as the tertiary. Under that ordering, the recursive equations above are always moving strictly downward, toward the minimum element $C = \{\}$, for which $c(C) = d(C) = e(C) = 0$. □

**Lemma 2.** *Case 4 detects and handles all type inconsistencies in the input, even those embedded within subexpressions.*

*Proof.* Recall that within any metaexpression, a metavariable can only appear as a leaf of the syntax tree. We view function application in the domain language (written $(f\ a_1\ \ldots\ a_n)$ after the manner of LISP) as a node with $n + 1$ children, the first of which is $f$, so even when metavariables are used as functions in the domain language, they are still leaves of syntax trees. Furthermore, in an expression function application $P(x)$, we see $P$ and $x$ as the two children of an expression function application, meaning that $P$ is still in leaf position. Thus metavariables are always leaves.

In a constraint set $C$, metavariables can only appear in the left hand side of constraints. If any left hand side of any constraint in $C$ is non-atomic and not an expression function application, cases 1 through 3 break it into its children before case 4 is considered. In the remainder of this proof, we assume that the algorithm has reached case 4 or some later case, and thus the only nonatomic left hand sides remaining in $C$ are expression function applications.

Even one occurrence of a metavariable $m$ in a constraint determines whether $m$ must be instantiated as an expression or as an expression function. Thus to create a type inconsistency requires exactly two occurrences of $m$ in a left hand side of a constraint. One occurrence must be as the first child of an expression function application, and the other must not. These two occurrences may be in the left hand sides of two different constraints, or they may both be in the left hand side of a single constraint. We consider both cases.

Case (a), when the two occurrences of $m$ are in different constraints, $c_1$ and $c_2$: As stated above, the only non-atomic metaexpressions in $C$ are expression function applications. Let $c_1$ be the constraint in which $m$ appears as the first child of an expression function application. Because Definition 10 forbids nesting expression function applications, $c_1$ must be of the form $(m(t), e)$ for some $e \in E$ and $t \in E'$. Thus we know the form of $c_1$, so we turn to consider $c_2$. Let $L$ stand for its left hand side, and we find two subcases.

Subcase (i), when $L$ is atomic: For $m$ to appear in $L$, $m$ must be equal to $L$. This subcase is handled by the first equation in case 4 of the specification.

Subcase (ii), when $L$ is non-atomic: Then $c_2$ must be an expression function application. In order for $c_1$ and $c_2$ to create a type inconsistency, the $m$ must appear somewhere other than as the first child of $L$ (although it may also appear there). If it appears anywhere within the second child of $L$ (which need not be atomic), it will not be as an expression function, because Definition 10 forbids nested expression function applications. This subcase is handled by the second equation in case 4 of the specification.

Case (b), when the two occurrences of $m$ are in the same constraint, call it $c$: As stated above, the only non-atomic metaexpressions in $C$ are expression function applications. Since $m$ occurs twice in $c$, $c$ cannot be atomic, and thus must be an expression function application. Since Definition 10 forbids nested expression function applications, $c$ contains exactly one expression function application, itself. Thus $m$ must appear as the first child of $c$, and somewhere within (or equal to) the second child of $c$. Again by Definition 10, the occurrence of $m$ within $c$'s second child cannot be as an expression function, which is where we got the type inconsistency. This case is handled by the final equation in case 4 of the specification.

Thus every kind of type inconsistency that may exist in a constraint set $C$ will be exposed by cases 1 through 3 of the specification for case 4 to handle in one of its three equations. □

**Theorem 2.** *The above specification for $F$ is consistent with the original definition of $F$, Definition 17.*

*Proof.* We prove that each case in the specification is consistent with the original definition of $F$.

**Case 1, the base case:**

We must prove that when $C = \{\}$, $F(C, s) = \{s\}$. The set $F(C, s)$ is the set of minimal complete solutions of $C \cup \{s\}$, and in this case $C \cup \{s\} = \{s\}$. A solution $s'$ is a complete solution for $\{s\}$ iff $s' \supseteq s$, and a minimal solution iff $s' \not\supset s$. Thus $s'$ is a complete, minimal solution iff $s' = s$. So the set of complete, minimal solutions to $\{s\}$ is precisely $\{s\}$.

**Case 2, the atomic case:**

Let $C = C' \cup \{(a, e)\}$ for some $a, e \in E$ with $a$ atomic. We must prove that if $a = e$ then $F(C, s) = F(C', s)$, but if $a \neq e$ then $F(C, s) = \{\}$.

For the first half of our goal, assume that $a = e$. Any potential solution $s'$ satisfies $s'(a) = a$, because $a$ contains no metavariables. Thus $s'(a) = e$, meaning that all potential solutions obey the constraint $(a, e)$. Thus the constraint $(a, e)$ can be omitted from any constraint set without changing the results of $F$, that is, $F(C, s) = F(C', s)$.

The second half of the goal is a direct consequence of the observation that $\forall s', s'(a) = a$. If $a \neq e$ then $\forall s', s'(a) \neq e$, and thus no potential solution can be a complete solution for $C$, so $F(C, s) = \{\}$.

**Case 3, the non-atomic case:**

Let $C = C' \cup \{(t, e)\}$ for some non-atomic, non-expression-function-application $t \in E'$ and $e \in E$. Let $t$ have children $t_1, \ldots, t_n$. We must prove that if $t$ and $e$ have the same type and the same number of children, with $e$'s children being $e_1, \ldots, e_n$, then $F(C, s) = F(C' \cup \{(t_1, e_1), \ldots, (t_n, e_n)\}, s)$, but if not then $F(C, s) = \{\}$.

Assume that $t$ and $e$ have the same type and the same number of children, $t_1, \ldots, t_n$ and $e_1, \ldots, e_n$. Let $s'$ be any potential solution. Since $t$ is not a metavariable nor an expression function application, substitution distributes over its children, meaning that $s'(t)$ is an expression of the same type as $t$, but with children $s'(t_1), \ldots, s'(t_n)$. By Definition 5 and our assumptions, we have that $s'(t) = e$ iff $s'(t_1) = e_1 \wedge \ldots \wedge s'(t_n) = e_n$. Consequently, a potential solution satisfies the constraint $(t, e)$ iff it satisfies the constraints $(t_1, e_1)$ through $(t_n, e_n)$. Thus $F(C, s) = F(C' \cup \{(t_1, e_1), \ldots, (t_n, e_n)\}, s)$.

Assume that $t$ and $e$ either have different types or different numbers of children. Take any potential solution $s'$. As in the previous paragraph, because $t$ is not a metavariable nor an expression function application, we know that $s'(t)$ is an expression of the same type as $t$ but with children $s'(t_1)$ through $s'(t_n)$. Thus $s'(t)$ and $e$ either have different types or different numbers of children. Thus they are not equal, and so $s'$ does not satisfy the constraint $(t, e)$. Since $s'$ was arbitrary, no potential solution can satisfy it, and since $(t, e) \in C$, $F(C, s) = \{\}$.

**Case 4, the inconsistent case:**

Claim: Any constraint set with a type inconsistency has no minimal, complete solutions.

Proof of claim: Consider an arbitrary constraint set $C$ with a type inconsistency in it regarding the metavariable $m$. As in Lemma 2, $C$ must therefore somewhere contain an occurrence of $m$ as an expression function, and another occurrence of $m$ not as an expression function. Say the first occurrence is in a constraint $(c_1, c_2)$, and the second in a constraint $(c_3, c_4)$ (which may or may not be the same constraint, as discussed in Lemma 2).

Now take an arbitrary potential solution $s'$ of $C$. If $m$ is not in $s'$ then $s'$ is not a complete solution for $C$. If $s'(m)$ is an expression function, then the constraint $(c_3, c_4)$ is not satisfied by $s'$ because we cannot have $s'(c_3) = c_4$ since $c_4$ is an expression but $s'(c_3)$ is undefined, as it attempts to substitute a non-expression in as a subtree of an expression. If $s'(m)$ is not an expression function, then the constraint $(c_1, c_2)$ is not satisfied, because $s'(c_1) = c_2$ cannot hold since $c_2$ is an expression but $s'(c_1)$ is undefined, as it attempts to apply a non-expression-function as if it were an expression function. Thus at least one constraint in $C$ is violated by any $s'$, and so there are no complete solutions for $C$, and the claim is proven.

Returning to the business of proving Case 4, let $m_1, m_2 \in M$, $t_1, t_2 \in E'$, and $e_1, e_2 \in E$, as in the specification. Consider three subcases.

In the first subcase, let $C = C' \cup \{(m_1(t_1), e_1), (m_1, e_2)\}$. Clearly $C$ contains a type inconsistency, because in $(m_1(t_1), e_1)$, $m_1$ must be instantiated as an expression function, but in $(m_1, e_2)$, $m_1$ must be instantiated as the expression $e_2$. By the claim above, $C$ has no complete solutions, so $F(C, s) = \{\}$, as the first equation in Case 4 of the specification correctly computes.

In the second subcase, let $C = C' \cup \{(m_1(t_2), e_1), (m_2(t_1), e_2)\}, s)$ and assume that $m_1$ appears in $t_1$. Because $t_1$ appears properly inside an expression function application, and Definition 10 forbids nesting expression function applications, the occurrence of $m_1$ within it cannot be as an expression function. And yet in the constraint $(m_1(t_2), e_1)$ it appears as an expression function. Thus $C$ contains a type inconsistency, and the claim above again gives us that $F(C, s) = \{\}$. Thus the second equation in Case 4 of the specification is also correct.

In the final subcase, let $C = C' \cup \{(m_1(t_1), e_1)\}, s)$ and again assume that $m_1$ appears in $t_1$. As in the previous case, we have that $m_1$'s position in $m_1(t_1)$ is as an expression function, but is position within $t_1$ cannot be as an expression function. Thus $C$ contains a type inconsistency, the claim gives us that $F(C, s) = \{\}$, and we see that the final equation of Case 4 is correct.

**Case 5, the metavariable case:**

Let $C = C' \cup \{(m, e)\}$ for some $m \in M$ and $e \in E$. We must prove that if $m$ is in $s$ and $s(m) \neq e$ then $F(C, s) = \{\}$, but in all other cases, $F(C, s) = F(C', s \cup \{(m, e)\})$.

Assume that $m$ is in $s$ and $s(m) \neq e$. Any potential solution $s'$ for $C \cup s$ must extend $s$, and thus must satisfy $s'(m) = s(m) \neq e$. Since $(m, e) \in C$, we see that $s'$ is not a complete solution for $C$, and thus $s' \notin F(C, s)$. Because $s'$ was arbitrary, $F(C, s) = \{\}$.

Otherwise, consider any potential solution $s'$. Now $C \cup s = (C' \cup \{(m, e)\}) \cup s = C' \cup (s \cup \{(m, e)\})$, so by the comments after Definition 17, $s'$ is a complete solution for $(C, s)$ iff it is a complete solution for $(C', s \cup \{(m, e)\})$. Because the minimality of the solution set does not depend on the constraints that generated it, we therefore have that $F(C, s) = F(C', s \cup \{(m, e)\})$. The only way this depends upon the assumption that $m$ is not in $s$ or $s(m) = e$ is that those assumptions guarantee that $s \cup \{(m, e)\}$ remains a partial function, and thus the expression $F(C', s \cup \{(m, e)\})$ is defined.

**Case 6, the first of two expression function application cases:**

Let $C = C' \cup \{(m(t_1), e_1), (m(t_2), e_2)\}$ for any $m \in M$, $e_1 \neq e_2 \in E$, and $t_1, t_2 \in E'$. Let $A$ and $A'$ be defined as in case 6 of the specification. Let $v$ be a new variable as in that same case.

We must prove that

$$F(C, s) = \min\left(\bigcup_{S \in A'} F(C' \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\}, s_S)\right),$$

with $a_S$ and $s_S$ as defined in case 6 of the specification. Recall that $A'$ is nonempty, and every

$S \in A'$ is nonempty as well, as explained in the specification.

We need prove only that any complete solution to $(C, s)$ is also a complete solution to some $(C' \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\}, s_S)$, as well as the converse, because Lemma 1 then gives us the desired result.

Consider first the right-to-left direction of the implication we aim to show. Let $S$ be an arbitrary element of $A'$. We must show that a complete solution to $(C' \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\}, s_S)$ is a complete solution to $(C, s)$. Recall that $s_S(m)$ is defined to be $\lambda v.e_1[S/v]$.

Let $s'$ be an arbitrary complete solution for $C' \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\} \cup s_S$. In particular, $s' \supseteq s_S \supseteq s$, and $s'(m) = \lambda v.e_1[S/v]$. We must show that $s'(m(t_1)) = e_1$ and $s'(m(t_2)) = e_2$.

$$\begin{aligned}
s'(m(t_1)) &= s'(m)(s'(t_1)) && \text{expression function application structure} \\
&= (\lambda v.e_1[S/v])(s'(t_1)) && \text{see above about } s'(m) \\
&= e_1[S/v][v = s'(t_1)] && \beta \text{ reduction} \\
&= e_1[S/s'(t_1)] && \text{simplify composition, with } v \text{ not in } e_1 \\
&= e_1[S/e_1[a_S]] && s' \text{ satisfies } (t_1, e_1[a_S]) \\
&= e_1 && S \in A, \text{ so it is uniform on } e_1
\end{aligned}$$

Thus we have $s'(m(t_1)) = e_1$, and $s'$ satisfies $(m(t_1), e_1)$.

A similar argument shows that $s'$ satisfies $(m(t_2), e_2)$, and thus all of $C \cup \{(m(t_1), e_1), (m(t_2), e_2)\} \cup s$. Thus a complete solution for any $(C' \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\}, s)$ is a complete solution for $(C, s)$.

Consider now the left-to-right direction of the implication; let $s'$ be a complete solution for $(C, s)$, or equivalently, for $C' \cup \{(m(t_1), e_1), (m(t_2), e_2)\} \cup s$. We must show that for some $S \in A'$, $s'$ is a complete solution for $C' \cup \{(t_1, e_1[a_S]), (t_2, e_2[a_S])\} \cup s_S$.

Say that $s'(m) = \lambda v.b$, where $v$ is a variable not appearing in $C$ or $s$, and $b$ is an expression that is the "body" of the function. (The actual variable $v$ is irrelevant; there are infinitely many $\alpha$-equivalent ways to write $m$ down, and we can choose one for which $v$ is "new.") We know that $s'(m)$ must have this form, because $s'$ is a complete solution for a constraint set in which $m$ appears as an expression function.

Now let $V$ be the set of addresses within $b$ of instances of $v$ (which are also the addresses within $s'(b)$ of instances of $v$, because $v$ does not appear in $s'$). We aim to show that $s'$ satisfies $C' \cup \{(t_1, e_1[a_V]), (t_2, e_2[a_V])\} \cup s_V$ and $V \in A'$. Showing $V \in A'$ means showing that $V$ is an ancestor of the difference set between $e_1$ and $e_2$, $V$ is uniform on $e_1$ and $e_2$, and that $s \cup \{(m, \lambda v.e_1[V/v])\}$ is a partial function. (These requirements come from the definition of $A'$ given in case 6 of Section 3.) Notice that the final requirement on the list is a consequence of the earlier facts we must show about $s'$, and thus we can skip that one.

Let us first show that $s'$ satisfies both $(t_1, e_1[a_V])$ and $(t_2, e_2[a_V])$. Because $a_V$ might be any element of $V$, let $a \in V$ be arbitrary, and we show that $s'$ satisfies $(t_1, e_1[a])$ and $(t_2, e_2[a])$. From the equations

$$\begin{aligned}
s'(m(t_1)) &= (s'(m))(s'(t_1)) && \text{expression function application structure} \\
&= (\lambda v.b)(s'(t_1)) && \text{form of } m \text{ given above} \\
&= b[v = s'(t_1)] && \beta\text{-reduction}
\end{aligned}$$

we have that $(s'(m(t_1)))[a] = b[w = s'(t_1)][a]$. Since $V$ is the addresses of every $v$ in $b$, and $a \in V$, we have that $(s'(m(t_1)))[a] = s'(t_1)$. Substituting $s'(m(t_1)) = e_1$ gives $e_1[a] = s'(t_1)$. A similar argument could show that $e_2[a] = s'(t_2)$, and so $s'$ satisfies $\{(t_1, e_1[a]), (t_2, e_2[a])\}$ for any $a \in V$, and in particular, for $a_V$.

We now show that $s'$ extends $s_V$. As defined in Section 3, $s_V = s \cup \{(m, \lambda v.e_1[V/v])\}$. Since we assumed that $s'$ is a complete solution for a superset of $s$, all that remains is to show that $s'(m) = \lambda v.e_1[V/v]$. We assumed that $s'(m) = \lambda v.b$, so we must show that $b = e_1[V/v]$. The equation chain above showed that $s'(m(t_1)) = b[v = s'(t_1)]$. We have assumed that $s'(m(t_1)) = e_1$, and we just proved that $s'(t_1) = e_1[a]$, so that equation becomes $e_1 = b[v = e_1[a]]$. Consequently, we have that $e_1[V/v] = b[v = e_1[a]][V/v]$, and the definition of $V$ means that the right hand side is simply $b$, as desired. (Although we did not need it in this paragraph, note that a similar argument also shows that $e_2[V/v] = b$, a fact we will use below.)

We now show that $V$ is an ancestor of the difference set between $e_1$ and $e_2$. Call the difference set $D$. By Definition 8, we must show that any address in $D$ has an ancestor in $V$ and every address in $V$ has a descendant in $D$. From the previous paragraph, we have that $e_1[V/v] = b = e_2[V/v]$.

First take an arbitrary $d \in D$. By Definition 6, $d$ is of maximal length, in the sense that the difference between $e_1[d]$ and $e_2[d]$ is observable at the address $d$, not requiring us to inspect proper subtrees. But because $d$ has no ancestor in $V$, the change from $e_1$ to $e_1[V/v]$ (and from $e_2$ to $e_2[V/v]$) alters only proper subtrees of $e_1[d]$ and $e_2[d]$ (if it even alters them at all). Consequently, we can still observe that $e_1[V/v] \neq e_2[V/v]$ by inspecting $e_1[V/v][d]$ and $e_2[V/v][d]$. This contradicts the fact that $e_1[V/v] = e_2[V/v]$, as desired.

Second take an arbitrary $a \in V$. We know that $b[v = s'(t_1)] = e_1 \neq e_2 = b[v = s'(t_2)]$, and so $s'(t_1) \neq s'(t_2)$. By the definition of $V$, $b[v = s'(t_1)][a] = s'(t_1)$ and $b[v = s'(t_2)][a] = s'(t_2)$, and by a few steps of substitution, $e_1[a] \neq e_2[a]$. By Definition 6, some $d \in D$ must be a descendant of $a$. This completes the second half of the proof that $V$ is an ancestor of the difference set between $e_1$ and $e_2$.

We now show that $V$ is uniform on $e_1$ and $e_2$. Recall from earlier that $e_1 = b[v = e_1[a]]$, which we can also write as $e_1 = b[V/e_1[a]]$, meaning that $V$ is uniform on $e_1$. A similar argument holds for $e_2$. We have thus shown all of our goals for the left-to-right direction, and thus have completed both directions, and thus completed this case of the proof.

**Case 7, the second of two expression function application cases:**

Let $C = C' \cup \{(m(t_1), e), (m(t_2), e)\}$ for any $m \in M$, $e \in E$, and $t_1, t_2 \in E'$. Let $U$ and $A_u$ be defined as in case 7 of the specification. Let $v$ be a new variable as in that same case.

We must prove that

$$F(C, s) = \min\left(\left(\bigcup_{u \in U, S \subseteq A_u, S \neq \emptyset} F(C' \cup \{(t_1, u), (t_2, u)\}, s_S)\right) \cup F(C', s \cup \{(m, \lambda v.e)\})\right).$$

We need only prove that any complete solution to $(C, s)$ is also a complete solution to some $(C' \cup \{(t_1, u), (t_2, u)\}, s_S)$ or to $(C', s \cup \{(m, \lambda v.e)\})$, as well as the converse, because Lemma 1 then gives us the desired result.

First we show the implication from left to right. Assume $s'$ is a complete solution to $C' \cup \{(m(t_1), e), (m(t_2), e)\} \cup s$. Obviously $m$ is in $s'$, so let's say $s'(m) = \lambda v.b$ (where $b$ just stands for the body of $m$). Note that $s'(m(t_1)) = (\lambda v.b)(s'(t_1)) = b[v = s'(t_1)] = e$, and the same holds for $t_2$.

If $v$ does not appear free in $b$ then $b = e$ by that equation, making $s'$ a complete solution for $C' \cup s \cup \{(m, \lambda v.e)\}$, which is the second disjunct on the right hand side of the desired implication. That disjunct is only included if $s \cup \{(m, \lambda v.e)\}$ is a partial function, but in this case it is, as a subset of $s'$. Thus in that case, the left-to-right implication is proven. So it remains to consider when $v$ appears free in $b$. Let $S'$ be the set of addresses in $b$ where $v$ appears.

Claim: $\exists u \in U, S' \subseteq A_u, S' \neq \emptyset$. Proof of claim: Let $a$ be the address of one of the occurrences of $v$ in $b$. Because $b[v = s'(t_1)] = e$, we have that $e[a] = s'(t_1)$. So let $u = s'(t_1)$ and thus $u \in U$ because $U$ is defined as the set of subexpressions of $e$. Now consider an arbitrary $a' \in S'$. Since $a'$, too, is an address of $v$ in $b$, we have that $e[a'] = s'(t_1)$ as well, and thus $a' \in A_u$. Because $a'$ was arbitrary, $S' \subseteq A_u$. Because $a \in S'$, we have $S' \neq \emptyset$, and the claim is proven. (The indexed union is also limited to sets $S$ for which $s_S$ is a partial function. The set $S'$ is such a set, because the next paragraph shows that $s_{S'} \subseteq s'$, and we have as a hypothesis that $s'$ is a partial function.)

Now we show that $s'$ is a complete solution to $C' \cup \{(t_1, u), (t_2, u)\} \cup s_{S'}$, with $u = s'(t_1)$ as in the previous paragraph. For $s'$ to satisfy $(t_1, u)$ means that $s'(t_1) = u$, which is simply the definition of

$u$. For $s'$ to satisfy $(t_2, u)$, we must have $s'(t_1) = s'(t_2)$. Because $s'(m(t_1)) = e = s'(m(t_2))$, either $s'(t_1) = s'(t_2)$ or $m$ is a constant function; the latter option is not the case because we know that $v$ appears free in $b$.

Recall that $s_{S'} = s \cup \{(m, \lambda v.e[S'/v])\}$. To show that $s'$ satisfies $s_{S'}$ means that we must show that $s'(m) = \lambda v.e[S'/v]$. We know that $s'(m) = \lambda v.b$, so we inspect the bodies to determine if $b = e[S'/v]$. Of course, $S'$ was defined to be the locations in $b$ where $v$ occurs, so clearly $b$ and $e[S'/v]$ occur at every address in $S'$. And because we have the equation $b[v = s'(t_1)] = e$ from earlier, we know that at every address not in $S'$, $b$ and $e$ are also equal. Thus $b = e[S'/v]$ and $s'$ satisfies $s_{S'}$. Thus $s'$ satisfies $C' \cup \{(t_1, u), (t_2, u)\} \cup s_{S'}$, and the left-to-right implication is proven.

Second we show the implication from right to left. Consider first the case when there is some $u \in U$ and $S \subseteq A_u$ such that $s'$ is a complete solution for $C' \cup \{(t_1, u), (t_2, u)\} \cup s_S$. Recall that $s'$ extending $s_S$ means that $s'(m) = \lambda v.e[S/v]$ by the definition of $s_S$.

$$\begin{aligned} s'(m(t_1)) &= s'(m)(s'(t_1)) && \text{expression function application structure} \\ &= (\lambda v.e[S/v])(s'(t_1)) && \text{fact above about } s' \\ &= e[S/v][v = s'(t_1)] && \beta \text{ reduction} \\ &= e[S/s'(t_1)] && \text{simplify composition, with } v \text{ not in } e \\ &= e[S/u] && s' \text{ satisfies } (t_1, u) \\ &= e && S \subseteq A_u\text{, plus the definition of } A_u \end{aligned}$$

A similar fact can be shown for $t_2$. Thus $s'$ is a complete solution for $C' \cup \{(m(t_1), e), m(t_2), e)\} \cup s$, which is the left hand side of the implication.

Consider second the case when $s'$ is a complete solution for $C' \cup s \cup \{(m, \lambda v.e)\}$. Then $s'(m) = \lambda v.e$, so clearly $s'(m(t_1)) = s'(m)(s'(t_1)) = (\lambda v.e)(s'(t_1)) = e$. The same holds for $t_2$. Thus $s'$ is a complete solution for $C' \cup \{(m(t_1), e), (m(t_2), e)\} \cup s$. Thus in both cases we have shown the right-to-left implication, completing the proof of this case.

**Case 8, into which all remaining possibilities fall:**

Recall that $C$ is nonempty and contains only constraints whose left hand sides are expression function applications, and no two such expression function applications begin with the same metavariable.

Consider first the case when $m$ appears in $s$. We wish to show that

$$F(C \cup \{(m(t), e)\}, s) = F(C \cup \{(s(m)(t), e)\}, s).$$

For the left-to-right inclusion, assume $s' \in F(C \cup \{(m(t), e)\}, s)$, meaning that $s'$ satisfies $C \cup s$ and $s'(m(t)) = e$. The inclusion holds by the following chain of equalities.

$$\begin{aligned} s'(s(m)(t)) &= s'(s(m))(s'(t)) && \text{expression function application structure} \\ &= s(m)(s'(t)) && s(m) \text{ contains no metavariables} \\ &= s'(m)(s'(t)) && s' \text{ extends } s \text{ and } m \text{ is in } s \\ &= s'(m(t)) && \text{expression function application structure} \\ &= e && \text{assumption above about } s' \end{aligned}$$

For the right-to-left inclusion, assume $s' \in F(C \cup \{(s(m)(t), e)\}, s)$, meaning that $s'$ satisfies $C \cup s$ and $s'(s(m)(t)) = e$. The inclusion holds by the following chain of equalities.

$$\begin{aligned} s'(m(t)) &= s'(m)(s'(t)) && \text{expression function application structure} \\ &= s(m)(s'(t)) && s' \text{ extends } s \text{ and } m \text{ is in } s \\ &= s'(s(m))(s'(t)) && s(m) \text{ contains no metavariables} \\ &= s'(s(m)(t)) && \text{expression function application structure} \\ &= e && \text{assumption above about } s' \end{aligned}$$

Thus both directions of the inclusion hold when $m$ is in $s$. (Technically we have ignored minimality in this case, just showing that complete solutions for each side are complete solutions for the other. One can then trivially invoke Lemma 1 with $n = 1$ to add the minimality component.)

Consider now the other case, when $m$ does not appear in $s$. We wish to show that

$$F(C \cup \{(m(t), e)\}, s) = \min\left(\left(\bigcup_{u \in U, S \subseteq A_u, S \neq \emptyset} F(C \cup \{(t, u)\}, s_S)\right) \cup F(C, s \cup \{(m, \lambda v.e)\})\right).$$

This equality is so similar to the equation from case 7 that its proof applies here almost entirely unchanged. For the sake of completeness and care, we include the proof here with the appropriate changes made, but it will seem very familiar.

We need only prove that any complete solution to $(C, s)$ is also a complete solution to some $(C' \cup \{(t_1, u), (t_2, u)\}, s_S)$ or to $(C', s \cup \{(m, \lambda v.e)\})$, as well as the converse, because Lemma 1 then gives us the desired result.

First we show the implication from left to right. Assume $s'$ is a complete solution to $C' \cup \{(m(t), e)\} \cup s$. Obviously $m$ is in $s'$, so let's say $s'(m) = \lambda v.b$ (where $b$ just stands for the body of $m$). Note that $s'(m(t)) = (\lambda v.b)(s'(t)) = b[v = s'(t)] = e$.

If $v$ does not appear free in $b$ then $b = e$ by that equation, making $s'$ a complete solution for $C' \cup s \cup \{(m, \lambda v.e)\}$, which is the second disjunct on the right hand side of the desired implication. That disjunct is only included if $s \cup \{(m, \lambda v.e)\}$ is a partial function, but in this case it is, as a subset of $s'$. Thus in that case, the left-to-right implication is proven. So it remains to consider when $v$ appears free in $b$. Let $S'$ be the set of addresses in $b$ where $v$ appears.

Claim: $\exists u \in U, S' \subseteq A_u, S' \neq \emptyset$. Proof of claim: Let $a$ be the address of one of the occurrences of $v$ in $b$. Because $b[v = s'(t)] = e$, we have that $e[a] = s'(t)$. So let $u = s'(t)$ and thus $u \in U$ because $U$ is defined as the set of subexpressions of $e$. Now consider an arbitrary $a' \in S'$. Since $a'$, too, is an address of $v$ in $b$, we have that $e[a'] = s'(t)$ as well, and thus $a' \in A_u$. Because $a'$ was arbitrary, $S' \subseteq A_u$. Because $a \in S'$, we have $S' \neq \emptyset$, and the claim is proven.

To see that $s'$ is a complete solution to $C' \cup \{(t, u)\} \cup s_{S'}$, with $u = s'(t)$ as in the previous paragraph, all that remains is to see that $s'$ satisfies $(t, u)$. In other words, we must verify that $s'(t) = u$ holds, but this is simply the definition of $u$.

Recall that $s_{S'} = s \cup \{(m, \lambda v.e[S'/v])\}$. To show that $s'$ satisfies $s_{S'}$ means that we must show that $s'(m) = \lambda v.e[S'/v]$. We know that $s'(m) = \lambda v.b$, so we inspect the bodies to determine if $b = e[S'/v]$. Of course, $S'$ was defined to be the locations in $b$ where $v$ occurs, so clearly $b$ and $e[S'/v]$ occur at every address in $S'$. And because we have the equation $b[v = s'(t)] = e$ from earlier, we know that at every address not in $S'$, $b$ and $e$ are also equal. Thus $b = e[S'/v]$ and $s'$ satisfies $s_{S'}$. Thus $s'$ satisfies $C' \cup \{(t, u)\} \cup s_{S'}$, and the left-to-right implication is proven.

Second we show the implication from right to left. Consider first the case when there is some $u \in U$ and $S \subseteq A_u$ such that $s'$ is a complete solution for $C' \cup \{(t, u)\} \cup s_S$. Recall that $s'$ extending $s_S$ means that $s'(m) = \lambda v.e[S/v]$ by the definition of $s_S$.

$$\begin{aligned}
s'(m(t)) &= s'(m)(s'(t)) && \text{expression function application structure} \\
&= (\lambda v.e[S/v])(s'(t)) && \text{fact above about } s' \\
&= e[S/v][v = s'(t)] && \beta \text{ reduction} \\
&= e[S/s'(t)] && \text{simplify composition, with } v \text{ not in } e \\
&= e[S/u] && s' \text{ satisfies } (t, u) \\
&= e && S \subseteq A_u, \text{ plus the definition of } A_u
\end{aligned}$$

Thus $s'$ is a complete solution for $C' \cup \{(m(t), e)\} \cup s$, which is the left hand side of the display formula.

Consider second the case when $s'$ is a complete solution for $C' \cup s \cup \{(m, \lambda v.e)\}$. Then $s'(m) = \lambda v.e$, so clearly $s'(m(t)) = s'(m)(s'(t)) = (\lambda v.e)(s'(t)) = e$. Thus $s'$ is a complete solution for $C' \cup \{(m(t), e)\} \cup s$. Thus in both cases we have shown the right-to-left implication, completing the proof of this case.

This completes the final equality proof for the final case, and thus completes the proof of the Theorem 2. □

# 5 Enumeration

My purpose for the matching algorithm documented herein is that it will most frequently be used to answer the question, "Does expression $e$ match pattern $p$?" Occasionally, a user may ask *how* an expression matches a pattern, which requires answering the question, "What is an example instantiation of $p$ that equals $e$?" The function $F$ defined in Section 3 is much more general. It always answers the question, "What is the complete set of instantiations of $p$ that equal $e$?"

This is required because the algorithm, while exploring potential solutions, does not know which will extend to be complete solutions, and thus must explore all avenues. In other words, the recursive nature of the problem requires computing all solutions down one branch of the recursion, because it does not know which, if any, will extend to full solutions when passed on down sibling branches of the same recursion.

And yet, at the same time, if the algorithm could terminate sooner with results sufficient to the caller's needs, then not doing so is wasteful. This is important because this algorithm may be run extremely often in some applications.

Consequently, this section discusses how to replace the definition of $F$ with a new function $N$, which will not compute all solutions at once, but will enumerate them as the caller iterates $N$. Here are the formal definitions.

**Definition 19** (computation state)**.** *A* computation state *is a triple $\sigma = (C, s, d)$, where $C$ is a constraint set, $s$ is a potential solution, and $d$ is any temporary data (for now just call it a set). The set of all computation states is written $\Sigma$. We also let $* \notin \Sigma$ signify an invalid computation state and define $\Sigma' = \Sigma \cup \{*\}$.*

We already know that we can turn a pattern-matching problem (match $p$ to $e$) into a trivial constraint set $\{(p, e)\}$. We can turn any constraint set $C$ into a trivial computation state $(C, \{\}, \{\})$.

**Definition 20** (initial computation state)**.** *A computation state $(C, \{\}, \{\})$ created from a constraint set $C$ is called an* initial computation state.

We will construct a function $N : \Sigma' \to \Sigma' \times ((M \times E) \cup \{*\})$, called the "next" function, which will enumerate all the complete solutions to a constraint set $C$. Given any $C$, we will construct its initial computation state $\sigma$ and iteratively apply $N$. Each application of $N$ will yield a new computation state as well as either a new solution $s \in M \times E$ or the $*$ object, which means that no additional solutions exist. We require that $N(*) = (*, *)$ and we use this as a flag to indicate when iteration of $N$ has reached a fixed point, and no further solutions are available.

An example may clarify this. Assume $C$ is a constraint set such that $F(C, \{\}) = \{s_1, s_2, s_3\}$. We can compute $F(C, \{\})$ using $N$ as follows.

1. Let $\sigma = (C, \{\}, \{\})$ be the initial computation state for $C$.

2. Compute $N(\sigma)$, and it should be $(\sigma_1, s_1)$ for some state $\sigma_1 \in \Sigma - \{\sigma\}$.

3. Compute $N(\sigma_1)$, and it should be $(\sigma_2, s_2)$ for some state $\sigma_2 \in \Sigma - \{\sigma, \sigma_1\}$.

4. Compute $N(\sigma_2)$, and it should be $(*, s_3)$, indicating that $s_3$ is the final solution that $N$ will yield for $C$. Alternately, if $N$ has not yet done enough computation to know that $s_3$ is the final solution, $N(\sigma_2)$ may yield $(\sigma_3, s_3)$, for some $\sigma_3 \in \Sigma - \{\sigma, \sigma_1, \sigma_2\}$.

5. Attempting to iterate $N$ further, with either $\sigma_3$ or $*$ as its input, would simply yield $(*, *)$, a fixed point.

Note that the three solutions $s_1, s_2, s_3$ are unordered, and although this example claims that they should arrive in the order $s_1, s_2, s_3$, that was just for simplicity of stating the example. The function $N$ may give the solutions in any order (but it will always give them in the same order, since it is a function).

If a caller wants to know all of $F(C, \{\})$, they can simply iterate $N$ enough times, and the total elapsed time of all those iterates should be comparable to the time it takes to compute $F$, because $N$ saves intermediate information in the data element $d$ of each computation state, so that it can "pick up right where it left off." But the main advantage is that if a caller only wants to know an example solution from among those in $F(C, \{\})$, or whether that set is nonempty, the caller can just do the first call of $N$, and stop there, saving significant time. In my intended application of this algorithm, the great majority of times that $F$ will need to be evaluated fall into this category, and thus using $N$ instead of $F$ may save significant time.

There is one important difference between the results of $F$ and $N$. Because $F$ is computing full solution sets before returning them, it can apply the min operator to ensure that only minimal solutions are returned. But $N$ returns solutions as soon as it finds them, pausing computation indefinitely. Thus it cannot apply the min operator, which requires a full solution set as input. Consequently, the definition of $N$ given below parallels the definition of $F$ *almost* perfectly; it differs by leaving out the uses of the min operator. Thus $N$ will occasionally return more solutions than $F$ will (but never less!).

We therefore propose the following recursive definition for $N$. Note that it essentially follows exactly the cases of $F$, except converts them into an enumeration rather than a complete computation. In each case, I write $N(C, s, d)$ as shorthand for $N(\sigma)$, where $\sigma = (C, s, d)$, in the way common to mathematics texts (rather than, for instance, insisting on $\sigma$'s status as a tuple, as in $N((C, s, d))$). Also, $N$ can be thought of as two component functions, $N_1 : \Sigma' \to \Sigma'$ and $N_2 : \Sigma' \to ((M \times E) \cup \{*\})$, a notation I make use of below.

1. **Base case:** $N(\{\}, s, d) = (*, s)$

2. **Atomic case:** If $a, e \in E$ and $a$ is atomic, then

$$N(C \cup \{(a, e)\}, s, d) = \begin{cases} N(C, s, d) & \text{if } a = e \\ & \\ (*, *) & \text{otherwise.} \end{cases}$$

3. **Non-atomic case:** If $a \in E$ and $t \in E'$ with $t$ non-atomic, not an expression function application, and having children $t_1, \ldots, t_n$, then

$$N(C \cup \{(t, e)\}, s, d) = \begin{cases} N(C \cup \{(t_1, e_1), \ldots, (t_n, e_n)\}, s, d) & \text{if } t, e \text{ have the same type and} \\ & \quad e \text{ has } n \text{ children, } e_1, \ldots, e_n \\ (*, *) & \text{otherwise.} \end{cases}$$

4. **Inconsistent case:** If $m_1, m_2 \in M$, $t_1, t_2 \in E'$, $e_1, e_2 \in E$, then

$$N(C \cup \{(m_1(t_1), e_1), (m_1, e_2)\}, s, d) = (*, *).$$

Furthermore, if $m_1$ appears in $t_1$, then

$$N(C \cup \{(m_1(t_2), e_1), (m_2(t_1), e_2)\}, s, d) = (*, *)$$

and

$$N(C \cup \{(m_1(t_1), e_1)\}, s, d) = (*, *).$$

5. **Metavariable case:** If $m \in M$ and $e \in E$, then
$$N(C \cup \{(m,e)\}, s, d) = \begin{cases} (*,*) & \text{if } m \text{ is in } s \text{ and } s(m) \neq e \\ \\ N(C, s \cup \{(m,e)\}, d) & \text{otherwise.} \end{cases}$$

6. **First of two expression function application cases:** If $m \in M$, $e_1 \neq e_2 \in E$, and $t_1, t_2 \in E'$, then define $A'$ as in Section 3, case 6. Call its elements $S_1$ through $S_k$ and fix an $a_{S_i}$ in each $S_i$ as in that same case. Then define $C_i = C \cup \{(t_1, e_1[a_{S_i}]), (t_2, e_2[a_{S_i}])$. Also, for each $S_i \in A'$, compute the corresponding $s_{S_i}$, using some new variable $v$ not in $C$ or $s$.

   TWe begin with a case that sets up the list of computation states over which we must compute a union.
$$N(C \cup \{(m(t_1), e_1), (m(t_2), e_2)\}, s, \{\}) = N(C, s, \{\langle 1, (C_1, s_{S_1}, \{\}), \ldots, (C_k, s_{S_k}, \{\})\rangle\})$$

   (The number 1 is used as the first element of the vector to distinguish this type of data from that used in other cases, below.)

   Next, we begin by enumerating all the solutions in the first entry of the union, if any, and moving on to the next entry if there are not any.
$$N(C, s, \{\langle 1, \sigma_1, \ldots, \sigma_k\rangle\}) =$$
$$\begin{cases} ((C, s, \{\langle 1, N_1(\sigma_1), \sigma_2, \ldots, \sigma_k\rangle\}), N_2(\sigma_1)) & \text{if } N_1(\sigma_1) \neq * \text{ and } N_2(\sigma_1) \neq * \\ \\ ((C, s, \{\langle 1, \sigma_2, \ldots, \sigma_k\rangle\}), N_2(\sigma_1)) & \text{if } N_1(\sigma_1) = * \text{ and } N_2(\sigma_1) \neq * \\ \\ N(C, s, \{\langle 1, \sigma_2, \ldots, \sigma_k\rangle\}) & \text{otherwise.} \end{cases}$$

   Finally, when the union has been exhausted, we yield the flag that says that the enumeration is over.
$$N(C, s, \{\langle 1\rangle\}) = (*,*)$$

7. **Second of two expression function application cases:** If $m \in M$, $e \in E$, and $t_1, t_2 \in E'$. Say $U = \{u_1, \ldots, u_k\}$, with $U$ as in Section 3 case 7. Let $v$ be a new variable not appearing in $C$ or $s$, so that we can speak of $s_S$ for any address set $S$. For each $u_k$, assume we enumerate all nonempty subsets $S \subseteq A_{u_k}$ for which $s_S$ is a partial function, and concatenate all such lists into one long list $\{S_1, \ldots, S_{k'}\}$. Then for each $i \in \{1, \ldots, k'\}$, fix some $a_i \in S_i$, define $C_i = C \cup \{(t_1, e[a_i]), (t_2, e[a_i])\}$. (Because $a_i \in S_i$, $e[a_i]$ will be the $u \in U$ for which $S_i \subseteq A_u$.)

   Then we begin with a case that sets up the list of computation states over which we must compute an indexed union.
$$N(C \cup \{(m(t_1), e), (m(t_2), e)\}, s, \{\}) = N(C, s, \{\langle 2, (C_1, s_{S_1}, \{\}), \ldots, (C_{k'}, s_{S'_k}, \{\})\rangle\})$$

   (The number 2 is used as a flag, just as 1 was in the previous case.)

   Next, we begin by enumerating all the solutions in the first entry of the indexed union, if any, and moving on to the next entry if there are not any.
$$N(C, s, \{\langle 2, \sigma_1, \ldots, \sigma_{k'}\rangle\}) =$$
$$\begin{cases} ((C, s, \{\langle 2, N_1(\sigma_1), \sigma_2, \ldots, \sigma_{k'}\rangle\}), N_2(\sigma_1)) & \text{if } N_1(\sigma_1) \neq * \text{ and } N_2(\sigma_1) \neq * \\ \\ ((C, s, \{\langle 2, \sigma_2, \ldots, \sigma_{k'}\rangle\}), N_2(\sigma_1)) & \text{if } N_1(\sigma_1) = * \text{ and } N_2(\sigma_1) \neq * \\ \\ N(C, s, \{\langle 2, \sigma_2, \ldots, \sigma_{k'}\rangle\}) & \text{otherwise.} \end{cases}$$

Finally, when the union has been exhausted, we fall back on the item outside the indexed union, which is a single recursive call.

$$N(C, s, \{\langle 2\rangle\}) = N(C, s \cup \{(m, \lambda v.e)\}, \{\}).$$

8. **Only remaining case:** The only case that remains is when $C \neq \emptyset$ contains only constraints with expression function applications on the left-hand side, with no expression function applied more than once. Consider a constraint $(m(t), e)$ and define $\{S_1, \ldots, S_{k'}\}$ and $s_{S_i}$ as in the previous case, but then let $C_i = C \cup \{(t, e[a_i])\}$. Then

$$N(C \cup \{(m(t), e)\}, s, \{\}) = \begin{cases} N(C \cup \{(s(m)(t), e)\}, s, \{\}) & \text{if } m \text{ is in } s \\ \\ N(C, s, \{\langle 2, (C_1, s_{S_1}, \{\}), \ldots, (C_{k'}, s_{S_{k'}}, \{\})\rangle\}) & \text{otherwise.} \end{cases}$$

   The recursion equations from case 6 will then process the union operation iteratively for us, just as they do in that case, suffixing the same final computation, as appropriate.

The similarity between the specification of $N$ in this section and that of $F$ in Section 3 make it so that a formal proof here that $N(C, s, \{\})$ enumerates $F(C, s)$ would be tedious, so it is not included here. Furthermore, recall the comments earlier in this section that $N$ occasionally enumerates non-minimal solutions to its inputs, meaning that $N$ enumerates either $F(C, s)$ or a superset thereof (but of course still containing only complete solutions to $(C, s)$).

# 6 Efficiency

In addition to the efficiency gains introduced by replacing a call to $F$ with one or more calls to $N$, there are other ways to make the algorithm from Section 3 faster. We mention a few here.

1. One of the slowest cases is case 8, which only occurs when a metavariable $m$ occurs precisely once in the original pattern, and there as an expression function. In many applications, this case may be avoidable. For instance, an application could reject rules of logic that fit this pattern, because the semantics of expression function application is not necessary for only one occurrence in a rule. In such a rule, the $P(x)$ (for the offending metavariable $P$) could be replaced simply by $P$.

2. Another case that can be improved is case 4, which takes $n^2$ time, with $n = |C|$, and will be run every time any case after 4 applies. But the same results could be accomplished by scanning the pattern once, before the algorithm is invoked, and verify that no metavariable $m$ occurs both in an expression function position and a non-expression-function position. Then case 4 can be dropped from the algorithm.

3. Cases should be ordered so that the earlier cases are both fast and likely to lead to an immediate result (not a recursive call, thus ending an entire subtree of the search space). Cases 1 through 5 all fit this description, but their order may not be optimal. Case 5 must appear after case 4, but if that case is removed (as per the previous item in this list) then case 5 could be moved earlier. It might fit best before case 3, because while both have a chance of terminating the search immediately (with a result of $\{\}$), the recursive half of case 5 is faster to evaluate than the recursive half of case 3.

4. Certain computations within the algorithm can be expensive. For instance, in case 7, enumerating all the subexpressions of $e$ may be an expensive operation. The results of such operations could be cached, so that future runs of the matching algorithm on similar inputs would be able to take advantage of the cache.

5. In parts 6 through 8 of the definition of $N$, a lengthy data vector $d$ is computed. Especially in the latter two cases, it can be exponential in size, and thus expensive to compute. Instead, permitting $d$ to be an enumerator that can be called to iteratively enumerate the data vector can be more efficient. This is a similar strategy to the construction of $N$ from $F$ in the first place.

# 7 Respecting Free Variables

It is easy to modify the definitions of $F$ or $N$ so that they only yield solutions that satisfy Definition 18. The thing to avoid is performing a substitution that causes variable capture (a formerly free variable becomes bound in the process of substitution).

Assume that we have a pattern $p$ to which we wish to match an expression $e$, and thus we will be constructing the constraint set $C = \{(p, e)\}$, and then calling either $F(C, \{\})$ or $N(C, \{\}, \{\})$.

Before invoking $F$ or $N$:

1. Find all subexpressions $b_1, \ldots, b_n$ of $p$ that are of binding type, and call their bound variables $v_1, \ldots, v_n$. For each $i \in \{1, \ldots, n\}$, compute the set of metavariables appearing free in $b_i$, and call them $m_{i,1}, \ldots, m_{i,k_i}$. Each pair $(v_i, m_{i,j})$ is a restriction on the solutions we are seeking, because the first half of Definition 18 says that we must keep only solutions for which $s(m_{i,j})$ is free in $b_i$, which is the same as saying that $s(v_i)$ does not appear free in $s(m_{i,j})$. Let the (finite) set of all such pairs be $R_1$.

2. Find all expression function applications $P_1(x_1), \ldots, P_m(x_m)$ in $p$. Each pair $(P_i, x_i)$ is also a restriction on the solutions we are seeking, because the second half of Definition 18 says that we must keep only solutions for which $s(m_i)$ is free to be plugged into the function $s(P_i)$, which is the same as saying that $s(m_i)$ is free to replace the variable of $s(P_i)$ in the body of $s(P_i)$. Let the (finite) set of all such pairs be $R_2$.

Assume that we have pre-computed $R_1$ and $R_2$ before invoking $F$ or $N$. When computing $F(C, \{\})$, only one modification needs to be made. In any case that adds a new pair to the potential solution $s$, add constraints to require that the newly extended potential solution not violate any restriction in $R_1$ or $R_2$. As an example, here is case 5, modified with such restrictions.

$$F(C \cup \{(m, e)\}, s) = \begin{cases} \{\} & \text{if } m \text{ is in } s \text{ and } s(m) \neq e, \\ & \quad \text{or if } \exists (r_1, r_2) \in R_1,\ s'(r_1) \text{ appears free in } s'(r_2), \\ & \quad \text{where } s' = s \cup \{(m, e)\} \\ & \quad \text{or if } \exists (r_1, r_2) \in R_2,\ s'(r_2) \text{ is not free to replace} \\ & \quad \text{the variable of } s'(r_1) \text{ in the body of } s'(r_1) \\ & \\ F(C, s \cup \{(m, e)\}) & \text{otherwise.} \end{cases}$$

A similar modification can be made to the relevant cases of the definition of $N$. Again, here is case 5 as an example.

$$N(C \cup \{(m, e)\}, s, d) = \begin{cases} (*, *) & \text{if } m \text{ is in } s \text{ and } s(m) \neq e, \\ & \quad \text{or if } \exists (r_1, r_2) \in R_1,\ s'(r_1) \text{ appears free in } s'(r_2), \\ & \quad \text{where } s' = s \cup \{(m, e)\} \\ & \quad \text{or if } \exists (r_1, r_2) \in R_2,\ s'(r_2) \text{ is not free to replace} \\ & \quad \text{the variable of } s'(r_1) \text{ in the body of } s'(r_1) \\ & \\ N(C, s \cup \{(m, e)\}, d) & \text{otherwise.} \end{cases}$$

One would need to make such changes to cases 6, 7, and 8, to prevent metavariable assignments that violate restrictions in $R_1$ or $R_2$. The point of this section is that these changes can be made without entirely reworking the algorithms in question; they are a simple add-on after the fact. In fact, such filters could be applied completely outside the algorithms themselves, thus leaving the algorithms unchanged (although less efficient by exploring a larger solution space at times).

# 8 Examples

This section contains examples of using the matching algorithm on common rules of first-order logic, with both correct and incorrect instances. This serves two purposes. First, it helps the reader understand $F$ and $N$ by seeing them used. Second, it is a second check (beyond the proof of Theorem 2) to verify that the specification in Section 3 is correct. I include here four examples, but many more could be created, and perhaps should.

In each of these examples, I assume the context of trying to match a purported use of a rule of logic to the definition of the rule. I construct rules with the symbol $\rho$ at the head, and presume that all variables in a rule are metavariables. Purported rule uses are also constructed with $\rho$ as the head symbol, so that they have the appropriate structure to attempt to match them against the rule itself. Each step in each computation shown below cites just a single number, the case from Section 3 that applies.

Note that sometimes there will be steps in which constraints should be added to $C$, but $C$ actually gets shorter. This is because the constraints are already in $C$, and since $C$ is a set, they do not need to be listed twice.

Recall that the earliest applicable case in Section 3 is always applied first. Thus an element listed "first" in $C$ (as written, below) may not be processed first by the algorithm. Of course, since $C$ is a set, it does not actually order its elements.

1. Assume that a user has justified $x^2+1=0 \wedge y=5$, with premises $x^2+1=0$ and $y=5$, from the conjunction introduction rule, which has the form $A$, $B$, therefore $A \wedge B$. We construct the pattern $p=\rho(A,B,A\wedge B)$ and the expression $e=\rho(x^2+1=0,y=5,x^2+1=0\wedge y=5)$ and compute $F(\{(p,e)\},\{\})$ as follows.

$$\begin{aligned}
&F(\{(\rho(A,B,A\wedge B),\rho(x^2+1=0,y=5,x^2+1=0\wedge y=5))\},\{\}) && \\
&= F(\{(\rho,\rho),(A,x^2+1=0),(B,y=5),(A\wedge B,x^2+1=0\wedge y=5)\},\{\}) && 3\\
&= F(\{(A,x^2+1=0),(B,y=5),(A\wedge B,x^2+1=0\wedge y=5)\},\{\}) && 2\\
&= F(\{(A,x^2+1=0),(B,y=5),(\wedge,\wedge)\},\{\}) && 3\\
&= F(\{(A,x^2+1=0),(B,y=5)\},\{\}) && 2\\
&= F(\{(B,y=5)\},\{(A,x^2+1=0)\}) && 5\\
&= F(\{\},\{(A,x^2+1=0),(B,y=5)\}) && 5\\
&= \Big\{\{(A,x^2+1=0),(B,y=5)\}\Big\} && 1
\end{aligned}$$

The algorithm has correctly determined that the instantiation of $A$ as $x^2+1=0$ and $B$ as $y=5$ is the only way for the expression to match the pattern.

2. Assume that a user has justified $(-0.1)^4 \geq 0$, with premise $\forall x.x^4 \geq 0$, from the universal elimination rule, which has the form $\forall x.P(x)$ therefore $P(t)$. We construct the pattern $p=\rho(\forall x.P(x),P(t))$ and the expression $e=\rho(\forall x.x^4\geq 0,(-0.1)^4\geq 0)$ and compute $F(\{(p,e)\},\{\})$

as follows.

$$\begin{aligned}
&F(\{(\rho(\forall x.P(x), P(t)), \rho(\forall x.x^4 \geq 0, (-0.1)^4 \geq 0)\}, \{\}) && \\
&= F(\{(\rho, \rho), (\forall x.P(x), \forall x.x^4 \geq 0), (P(t), (-0.1)^4 \geq 0)\}, \{\}) && 3 \\
&= F(\{(\forall x.P(x), \forall x.x^4 \geq 0), (P(t), (-0.1)^4 \geq 0)\}, \{\}) && 2 \\
&= F(\{(\forall, \forall), (x, x), (P(x), x^4 \geq 0), (P(t), (-0.1)^4 \geq 0)\}, \{\}) && 3 \\
&= F(\{(x, x), (P(x), x^4 \geq 0), (P(t), (-0.1)^4 \geq 0)\}, \{\}) && 2 \\
&= F(\{(P(x), x^4 \geq 0), (P(t), (-0.1)^4 \geq 0)\}, \{(x, x)\}) && 5 \\
&= F(\{(x, x), (t, -0.1)\}, \{(x, x), (P, \lambda v.v^4 \geq 0)\}) && \\
&\qquad \cup\ F(\{(x, x^4), (t, (-0.1)^4)\}, \{(x, x), (P, \lambda v.v \geq 0)\}) && \\
&\qquad\quad \cup\ F(\{(x, x^4 \geq 0), (t, (-0.1)^4 \geq 0)\}, \{(x, x), (P, \lambda v.v)\}) && 6 \\
&= F(\{(t, -0.1)\}, \{(x, x), (P, \lambda v.v^4 \geq 0)\}) \cup \{\} \cup \{\} && 5 \text{ (3 times)} \\
&= F(\{\}, \{(x, x), (P, \lambda v.v^4 \geq 0), (t, -0.1)\}) && 5 \\
&= \Big\{\{(x, x), (P, \lambda v.v^4 \geq 0), (t, -0.1)\}\Big\} && 1
\end{aligned}$$

The algorithm has correctly determined that the instantiation of $x$ as $x$, $P$ as the expression function taking $v$ to $v^4 \geq 0$, and $t$ as $-0.1$ is the only way for the expression to match the pattern.

3. Assume that a user has justified $m|j$, with premises $\exists n.n|3k-2$ and $\forall m.m|3k-2 \to m|j$, from the existential elimination rule, which has the form $\exists x.P(x)$ and $\forall y.P(y) \to Q$ therefore $Q$. We construct the pattern $p = \rho(\exists x.P(x), \forall y.P(y) \to Q, Q)$ and the expression $e = \rho(\exists n.n|3k-2, \forall m.m|3k-2 \to m|j, m|j)$ and compute $F(\{(p, e)\}, \{\})$ as follows.

$$\begin{aligned}
&F(\{(\rho(\exists x.P(x), \forall y.P(y) \to Q, Q), \rho(\exists n.n|3k-2, \forall m.m|3k-2 \to m|j, m|j))\}, \{\}) && \\
&= F(\{(\rho, \rho), (\exists x.P(x), \exists n.n|3k-2), (\forall y.P(y) \to Q, \forall m.m|3k-2 \to m|j), (Q, m|j)\}, \{\}) && 3 \\
&= F(\{(\exists x.P(x), \exists n.n|3k-2), (\forall y.P(y) \to Q, \forall m.m|3k-2 \to m|j), (Q, m|j)\}, \{\}) && 2 \\
&= F(\{(\exists, \exists), (x, n), (P(x), n|3k-2), (\forall y.P(y) \to Q, \forall m.m|3k-2 \to m|j), (Q, m|j)\}, \{\}) && 3 \\
&= F(\{(x, n), (P(x), n|3k-2), (\forall y.P(y) \to Q, \forall m.m|3k-2 \to m|j), (Q, m|j)\}, \{\}) && 2 \\
&= F(\{(x, n), (P(x), n|3k-2), (\forall, \forall), (y, m), (P(y) \to Q, m|3k-2 \to m|j), (Q, m|j)\}, \{\}) && 3 \\
&= F(\{(x, n), (P(x), n|3k-2), (y, m), (P(y) \to Q, m|3k-2 \to m|j), (Q, m|j)\}, \{\}) && 2 \\
&= F(\{(x, n), (P(x), n|3k-2), (y, m), (\to, \to), (P(y), m|3k-2), (Q, m|j)\}, \{\}) && 3 \\
&= F(\{(x, n), (P(x), n|3k-2), (y, m), (P(y), m|3k-2), (Q, m|j)\}, \{\}) && 2 \\
&= F(\{(P(x), n|3k-2), (y, m), (P(y), m|3k-2), (Q, m|j)\}, \{(x, n)\}) && 5 \\
&= F(\{(P(x), n|3k-2), (P(y), m|3k-2), (Q, m|j)\}, \{(x, n), (y, m)\}) && 5 \\
&= F(\{(P(x), n|3k-2), (P(y), m|3k-2)\}, \{(x, n), (y, m), (Q, m|j)\}) && 5 \\
&= F(\{(x, n), (y, m)\}, \{(x, n), (y, m), (Q, m|j), (P, \lambda v.v|3k-2)\}) && \\
&\qquad \cup\ F(\{(x, n|3k-2), (y, m|3k-2)\}, \{(x, n), (y, m), (Q, m|j), (P, \lambda v.v)\}) && 6 \\
&= F(\{(y, m)\}, \{(x, n), (y, m), (Q, m|j), (P, \lambda v.v|3k-2)\}) \cup \{\} && 5 \text{ (twice)} \\
&= F(\{\}, \{(x, n), (y, m), (Q, m|j), (P, \lambda v.v|3k-2)\}) && 5 \\
&= \Big\{\{(x, n), (y, m), (Q, m|j), (P, \lambda v.v|3k-2)\}\Big\} && 1
\end{aligned}$$

The algorithm has *incorrectly* identified the instantiation of $x$ as $n$, $y$ as $m$, $Q$ as $m|j$, and $P$ as the function taking $v$ to $v|3k-2$ as a solution. In reality, this solution does not respect free variables. But if we apply the strategy from Section 7, we can easily filter it out, as described in that section.

Enumerating the binding expressions in $p$ gives us $b_1 = \exists x.P(x)$ and $b_2 = \forall y.P(y) \to Q$. Enumerating the free metavariables in each gives none for $b_1$ and just $m_{1,1} = Q$ for $b_2$. This gives us the one restriction that for any potential solution $s$, we cannot have $s(y)$ appearing free in $s(Q)$. In this case, $s(y) = m$ and $s(Q) = m|j$, so we see that the restriction is violated, and this solution would be filtered out.

4. Assume that a user has justified $k^2 = 7^2$, with premises $k = 7$ and $7^2 = k^2$, from the equality elimination rule, which has the form $a = b$ and $P(a)$ therefore $P(b)$. We construct the pattern $p = \rho(a = b, P(a), P(b))$ and the expression $e = \rho(k = 7, 7^2 = k^2, k^2 = 7^2)$ and compute $F(\{(p, e)\}, \{\})$ as follows.

$$\begin{array}{ll}
\phantom{=}\ F(\{(\rho(a = b, P(a), P(b)), \rho(k = 7, 7^2 = k^2, k^2 = 7^2))\}, \{\}) & \\
= F(\{(\rho, \rho), (a = b, k = 7), (P(a), 7^2 = k^2), (P(b), k^2 = 7^2)\}, \{\}) & 3 \\
= F(\{(a = b, k = 7), (P(a), 7^2 = k^2), (P(b), k^2 = 7^2)\}, \{\}) & 2 \\
= F(\{(=, =), (a, k), (b, 7), (P(a), 7^2 = k^2), (P(b), k^2 = 7^2)\}, \{\}) & 3 \\
= F(\{(a, k), (b, 7), (P(a), 7^2 = k^2), (P(b), k^2 = 7^2)\}, \{\}) & 2 \\
= F(\{(b, 7), (P(a), 7^2 = k^2), (P(b), k^2 = 7^2)\}, \{(a, k)\}) & 5 \\
= F(\{(P(a), 7^2 = k^2), (P(b), k^2 = 7^2)\}, \{(a, k), (b, 7)\}) & 5 \\
= \min(F(\{(a, 7^2 = k^2), (b, k^2 = 7^2)\}, \{(a, k), (b, 7), (P, \lambda v.v)\})) & 6 \\
= \min(\{\})) = \{\} & 5
\end{array}$$

The algorithm has correctly determined that a single application of the equality elimination rule cannot accomplish what the user was attempting. Although it can be accomplished with several uses of the equality elimination rule, which must appear in both of its forms (both the form that concludes $P(b)$ from $P(a)$ and the form that concludes $P(a)$ from $P(b)$), this single-step attempt should be judged an invalid use of the rule, as it is.

# 9 Supporting Substitution Notation

Substitution notation is the common notation from mathematical logic in which we write $P[x = y]$ to mean the expression $P$ after all free occurrences of the subexpression $x$ have been replaced by the subexpression $y$. It is possible to convert a limited subset of substitution notation rules into functional notation rules. Specifically, a rule in substitution notation must obey these restrictions.

1. No substitution expression can contain another (e.g., one cannot write $P[x = y][w = z]$, nor $P[x = f[u = v]]$).

2. No substitution expression can contain in its third child a metavariable that appears elsewhere as the first child of a substitution expression (e.g., one cannot write $P[x = Q]$ and elsewhere $Q[y = z]$).

3. If a substitution expression $P[x = y]$ appears in one part of the rule, and $P[x' = y']$ appears in another part of the rule, then we must have $x = x'$, and it must be a metavariable.

In such cases, we can convert substitution-notation-style rules into function-notation-style rules as follows.

Given a rule $R$ in substitution notation, let $P_1, \ldots, P_n$ be the metavariables appearing as the first children of substitution expressions within $R$. By the constraints given above, for every $P_i$, every occurrence of it must have the same metavariable as the left hand side of its substitution equation. Thus we can list them, $x_1, \ldots, x_n$. Construct a new rule $R'$ by replacing each $P_i[x_i = y]$ in $R$ with $P_i(y)$ and each occurrence of a $P_i$ outside a substitution expression with $P_i(x_i)$.

**Theorem 3.** *Consider a particular expression vector $e$ to be matched against the rule $R$, if possible. Consider instead the constraint set $C = \{(R', e)\}$, for which we have a matching algorithm defined in this paper. Define a function $c$ on the space of potential solutions, so $c : (M \to E) \to (M \to E)$, which converts solutions to $C$ back into solutions to the original matching problem, as follows.*

$$c(s)(m) = \begin{cases} s(P_i)(s(x_i)) & \text{if } m = P_i \text{ for some } P_i \\ s(m) & \text{otherwise.} \end{cases}$$

*Then for any potential solution $s$, $s$ is a complete solution to $C$ iff $c(s)$ is a complete solution to the original matching problem.*

This theorem is slightly ill-defined, because we have not yet defined what it means to have a complete solution to matching problems containing substitution notation, but we trust that the reader will bear with this small lack of formality in this optional section of the paper.

*Proof.* Let $s$ be a potential solution.

Left-to-right implication: Assume $s$ is a complete solution to $C$. We aim to show that $c(s)$ is a complete solution to the original matching problem. For any address $a$ into $R$, if $R[a]$ contains no $P_i$, then

$$\begin{aligned} c(s)(R[a]) &= c(s)(R'[a]) && \text{because } R[a] = R'[a] \\ &= s(R'[a]) && \text{because } s \text{ and } c(s) \text{ agree on inputs without } P_i \\ &= e[a] && \text{because } s \text{ is a complete solution to } C \end{aligned}$$

So our concern is with the $P_i$ within $R$. They occur in two ways.

1. Consider any address $a$ of any expression $P_i[x_i = y]$ in $R$. Consequently, $R'[a]$ will be $P_i(y)$. We then have the following chain of equalities.

$$\begin{aligned} c(s)(R[a]) &= c(s)(P_i[x_i = y]) && R[a] = P_i[x_i = y] \\ &= c(s)(P_i)[c(s)(x_i) = c(s)(y)] && \text{substitutions commute} \\ &= c(s)(P_i)[s(x_i) = s(y)] && x_i \text{ and } y \text{ contain no } P_i \\ &= s(P_i)(s(x_i))[s(x_i) = s(y)] && \text{definition of } c \\ &= s(P_i)(s(y)) && \text{apply substitution} \\ &= s(P_i(y)) && \text{expression function application structure} \\ &= s(R'[a]) && R'[a] = P_i(y) \\ &= e[a] && s \text{ is a complete solution to } C \end{aligned}$$

2. Consider any address $a$ of any expression $P_i$ in $R$ that is not the first child of a substitution expression. Consequently, $R'[a]$ will be $P_i(x_i)$. We then have the following chain of equalities.

$$\begin{aligned} c(s)(R[a]) &= c(s)(P_i) && R[a] = P_i \\ &= s(P_i)(s(x_i)) && \text{definition of } c \\ &= s(P_i(x_i)) && \text{expression function application structure} \\ &= s(R'[a]) && R'[a] = P_i(x_i) \\ &= e[a] && s \text{ is a complete solution to } C \end{aligned}$$

Thus for any address $a$ into $R$, $c(s)(R[a]) = e[a]$, so $c(s)(R) = e$, making $c(s)$ a complete solution for the original problem.

Right-to-left implication: Now assume $c(s)$ is a complete solution to the original problem. We aim to show that $s$ is a complete solution to $C$. For any address $a$ into $R'$, if $R'[a]$ contains no expression function applications, then it contains no $P_i$ (since both are introduced to $R'$ at the same addresses). Thus $R[a]$ was not modified in the creation of $R'$, and $R'[a] = R[a]$.

$$\begin{aligned} s(R'[a]) &= c(s)(R'[a]) && s \text{ and } c(s) \text{ agree on inputs without } P_i \\ &= c(s)(R[a]) && R[a] = R'[a] \\ &= e[a] && c(s) \text{ is a complete solution to the original problem} \end{aligned}$$

So our concern is with the expression function applications in $R'$. Consider any address $a$ of any expression function application in $R'$. They come in two varieties.

1. Consider the case when $R'[a] = P_i(x_i)$ for some $i$. Such a case implies that $R[a] = P_i$.

$$\begin{aligned} s(R'[a]) &= s(P_i(x_i)) && R'[a] = P_i(x_i) \\ &= s(P_i)(s(x_i)) && \text{expression function application structure} \\ &= c(s)(P_i) && \text{definition of } c \\ &= c(s)(R[a]) && R[a] = P_i \\ &= e[a] && c(s) \text{ is a complete solution to the original problem} \end{aligned}$$

2. Consider the case when $R'[a] = P_i(y)$ for some $i$, with $y \neq x_i$. Such a case implies that $R[a] = P[x_i = y]$.

$$\begin{aligned} s(R'[a]) &= s(P_i(y)) && R'[a] = P_i(y) \\ &= s(P_i)(s(y)) && \text{expression function application structure} \\ &= s(P_i)(s(x_i))[s(x_i) = s(y)] && \text{apply substitution} \\ &= c(s)(P_i)[s(x_i) = s(y)] && \text{definition of } c \\ &= c(s)(P_i)[c(s)(x_i) = c(s)(y)] && s \text{ and } c(s) \text{ agree on expressions that contain no } P_i \\ &= c(s)(P_i[x_i = y]) && \text{substitutions commute} \\ &= c(s)(R[a]) && R[a] = P_i[x_i = y] \\ &= e[a] && c(s) \text{ is a complete solution to the original problem} \end{aligned}$$

Thus for any address $a$ into $R'$, $s(R'[a]) = e[a]$, so $s(R') = e$, making $s$ a complete solution for $C$. □

Consequently, if one is willing to accept the restrictions given at the start of this section, one can use substitution notation, and the matching algorithm can still be used. The conversion from $R$ to $R'$ needs to be used before matching, and the function $c$ defined in Theorem 3 needs to be applied after matching. Then Theorem 3 guarantees that complete solutions are reported correctly by that three-step procedure.

## References

[1] http://www.openmath.org