The Semantics of K

Formal Systems Laboratory
University of Illinois

August 21, 2017

Please feel free to contribute to this report in all ways. You could add new contents, remove redundant ones, refactor and organize the texts, and correct typos, but please follow the FSL rules for editing, though; e.g., <80 characters per line, each sentence on a new line, etc.

1 Matching Logic

Let us recall the basic grammar of matching logic from [**?**]. Assume a matching logic *signature* (S, Σ), and let Var_s be a countable set of *variables* of sort s. For simplicity, here we assume that the sets of *sorts* S and of *symbols* Σ are finite. We partition Σ in sets of symbols $\Sigma_{s_1 \ldots s_n, s}$ of *arity* $s_1 \ldots s_n, s$, where $s_1, \ldots, s_n, s \in S$. Then *patterns* of sort $s \in S$ are generated by the following grammar:

Add references.

$$
\begin{aligned}
\varphi_s ::= {} & x{:}s \quad \text{where } x \in Var \\
& \mid \varphi_s \wedge \varphi_s \\
& \mid \neg \varphi_s \\
& \mid \exists x{:}s'.\varphi_s \quad \text{where } x \in N \text{ and } s' \in S \\
& \mid \sigma(\varphi_{s_1}, \ldots, \varphi_{s_n}) \quad \text{where } \sigma \in \Sigma \text{ has } n \text{ arguments, and } \ldots
\end{aligned}
$$

The grammar above only defines the syntax of (well-formed) patterns of sort s. It says nothing about their semantics. For example, patterns $x{:}s \wedge y{:}s$ and $y{:}s \wedge x{:}s$ are distinct elements in the language of the grammar, in spite of them being semantically/provably equal in matching logic.

For notational convenience, we take the liberty to use mix-fix syntax for operators in Σ, parentheses for grouping, and omit variable sorts when understood. For example, if $Nat \in S$ and $\_+\_, \_*\_ \in \Sigma_{Nat \times Nat, Nat}$ then we may write $(x + y) * z$ instead of $\_*\_(\_+\_(x{:}Nat, y{:}Nat), z{:}Nat)$.

A matching logic *theory* is a triple (S, Σ, A) where (S, Σ) is a signature and A is a set of patterns called *axioms*. Like in many logics, sets of patterns may

I think we also need to talk about: other logical connectives as derived, free variables, capture-free substitution, equality. Add more as we need them.

be presented as *schemas* making use of meta-variables ranging over patterns, sometimes constrained to subsets of patterns using side conditions. For example:

$\varphi[\varphi_1/x] \wedge (\varphi_1 = \varphi_2) \rightarrow \varphi[\varphi_2/x]$	where φ is any pattern and φ_1, φ_2 are any patterns of same sort as x
$(\lambda x.\varphi)\varphi' = \varphi[\varphi'/x]$	where φ, φ' are *syntactic patterns*, that is, ones formed only with variables and symbols This is not true. Pattern φ contains quantifiers.
$\varphi_1 + \varphi_2 = \varphi_1 +_{Nat} \varphi_2$	where φ, φ' are *ground* syntactic patterns of sort *Nat*, that is, patterns built only with symbols `zero` and `succ`
$(\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi[\varphi_1/x] \rightarrow \varphi[\varphi_2/x])$	where φ is a *positive context in* x, that is, a pattern containing only one occurrence of x with no negation ($\neg$) on the path to x, and where φ_1, φ_2 are any patterns having the same sort

One of the major goals of this paper is to propose a formal language and an implementation, that allows us to write such pattern schemas.

2 A Calculus of Matching Logic

In this section, we propose a calculus of matching logic as a matching logic theory.

> Many people have developed calculi for mathematical reasoning. A calculus of logics is often called a *logical framework*. I prefer to speak of a *meta-logic* and its *object-logic*.
> By L. Paulson, *The Foundation of a Generic Theorem Prover*

In this proposal, the *object-logic* refers to matching logic, and we propose to use matching logic itself as the *meta-logic*[1]. The calculus of matching logic, denoted as $K = (S_K, \Sigma_K, A_K)$, is a matching logic theory where S_K, Σ_K, and A_K are sets of sorts, symbols, and axioms respectively., The main goal of this section is to present the calculus K, which mainly consists of built-in theories, abstract syntax trees (ASTs) of patterns, and matching logic proof system. They are introduced in details in the following separate sections.

2.1 Built-ins

Two matching logic theories, BOOL for boolean algebra and STRING for strings, are included in the calculus K. Their definitions have been introduced elsewhere (e.g. in [**?**]) and will not be discussed in this proposal. Sorts *Bool* and *String*

[1]Coq and Isabelle use fragments of higher-order logic as their meta-logics.

are included in S_K, and the following symbols are included in Σ_K with their usual axioms [?] added to A_K.

$$
\begin{aligned}
&\mathit{true}, \mathit{false} : \ \to \mathit{Bool} \\
&\mathit{not}\_ : \mathit{Bool} \to \mathit{Bool} \\
&\_\mathit{and}\_, \_\mathit{or}\_ : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool} \\
&\_\mathit{implies}\_ : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}
\end{aligned}
$$

Remark 1. Whenever we introduce a sort, say X, to S_K, we feel free to use *XList* as the sort of comma-separated lists over X without explicitly defining it.

2.2 Abstract Syntax Trees

The sort $\mathit{Sort} \in S_K$ is the sort of matching logic sorts, whose only constructor symbol[2] is $\#\mathit{sort} : \mathit{String} \to \mathit{Sort}$. The sort $\mathit{Symbol} \in S_K$ is the sort of matching logic symbols whose only constructor symbol is $\#\mathit{symbol} : \mathit{String} \times \mathit{SortList} \times \mathit{Sort} \to \mathit{Symbol}$. The sort $\mathit{Pattern} \in S_K$ is the sort for ASTs of patterns, with the following functional constructor symbols:

Xiaohong: Lucas points out that the #'s in the next constructors are confusing with the lifting function $\#\_$ that maps patterns to their ASTs.

$$
\begin{aligned}
&\#\mathit{variable} \colon \mathit{String} \times \mathit{Sort} \to \mathit{Pattern} \\
&\#\mathit{and}, \#\mathit{or}, \#\mathit{implies}, \#\mathit{iff} \colon \mathit{Pattern} \times \mathit{Pattern} \to \mathit{Pattern} \\
&\#\mathit{equals}, \#\mathit{contains} \colon \mathit{Pattern} \times \mathit{Pattern} \times \mathit{Sort} \to \mathit{Pattern} \\
&\#\mathit{not} \colon \mathit{Pattern} \to \mathit{Pattern} \\
&\#\mathit{exists}, \#\mathit{forall} \colon \mathit{String} \times \mathit{Sort} \times \mathit{Pattern} \to \mathit{Pattern} \\
&\#\mathit{application} \colon \mathit{Symbol} \times \mathit{PatternList} \to \mathit{Pattern} \\
&\#\mathit{top}, \#\mathit{bottom} \colon \mathit{Sort} \to \mathit{Pattern}
\end{aligned}
$$

There are also AST-related symbols included in Σ_K. For example, the symbol $\mathit{wellFormed} \colon \mathit{Pattern} \to \mathit{Bool}$ determines whether a pattern is well-formed (or more precisely, it determines whether an abstract syntax tree is a well-formed one of a pattern.) The symbol $\mathit{getSort} \in \Sigma_{\mathit{Pattern},\mathit{Sort}}$ takes a pattern and returns its sort. If the pattern is not well-formed, then *getSort* returns $\bot_{\mathit{Sort}}$; otherwise, *getSort* returns $\#\mathit{sort}(s)$ if the pattern has sort s. The symbol $\mathit{getFvs} \colon \mathit{Pattern} \to \mathit{PatternList}$ collects all free variables in a pattern. The symbol $\mathit{substitute} \colon \mathit{Pattern} \times \mathit{Pattern} \times \mathit{Pattern} \to \mathit{Pattern}$ takes a target pattern φ, a "find"-pattern ψ_1, and a "replace"-pattern ψ_2, and returns φ in which ψ_2 is substituted for ψ_1, denoted as $\varphi[\psi_2/\psi_1]$.

Side conditions can be defined as functional symbols from *Pattern* to *Bool*. For example, the symbol *syntactic* determines whether a pattern contains only

[2] A constructor symbol has its precise definition in matching logic. Please refer to [?].

variables and symbol applications. The symbol *ground* determines whether a pattern is variable-free, no matter free or bound. The symbol *groundSyntactic* determines whether a pattern is both syntactic and ground. They all can be easily defined in K.

Xiaohong: I want to introduce the (beta) axiom of lambda calculus and show how $\mathit{lambdaTerm}\colon \mathit{Pattern} \to \mathit{Bool}$ can be defined in K as a (final) example.

Example 2 (Lambda Calculus)**.** content...

2.3 Proof System

It is strongly recommended that readers read L. Paulson's *The Foundation of a Generic Theorem Prover*, especially Section 2, 3, and 4.

A proof system is a theorem generator. In K, the proof system of matching logic is captured by the functional symbol $\mathit{deducible}\colon \mathit{Pattern} \to \mathit{Bool}$, which returns *true* iff the argument pattern is a theorem. Given a matching logic pattern φ, we use $\#\varphi$ to denote its abstract syntax tree. The sharp $\#\_$ is called the *lifting function* that maps object-patterns to their meta-representations in K. It worths to point out that the lifting function *cannot* be defined in K no matter what. It is purely a mathematical notation and is not part of the calculus. A simple example is to consider $\#0$ and $\#(x-x)$, where $0 = x - x$ but their ASTs are different. This means that the lifting function does not allow equational substitution, which is a strong evidence that it is not part of the logic, because in matching logic we can prove the next theorem:

$$\psi_1 = \psi_2 \to \varphi[\psi_1] = \varphi[\psi_2].$$

We introduce the double bracket $[\![\_]\!]$, known as the semantics bracket, as follows:

$$[\![\varphi]\!] \equiv (\mathit{deducible}(\#\varphi) = \mathit{true})\,.$$

Intuitively, whenever there is an inference rule (axioms are considered as rules with zero premises)

$$\frac{\varphi_1, \ldots, \varphi_n}{\psi}$$

in matching logic, there is a corresponding axiom in K:

$$[\![\varphi_1]\!] \wedge \cdots \wedge [\![\varphi_n]\!] \to [\![\psi]\!].$$

Inference modulo theories can be considered in the same way. For any (syntactic) matching logic theory T whose axiom set is A, we can add

$$[\![\varphi]\!] \quad \text{for all } \varphi \in A$$

as axioms to K. We sometimes denote the extended theory as $\#T$ and call it the *meta-theory for T*.

2.4 Faithfulness

It remains a question whether the calculus K faithfully captures matching logic reasoning. The following definition of *faithfulness* is inspired by [**?**].

Definition 3. The calculus K is said to be faithful for matching logic, if for any matching logic syntactic theory T and its meta-theory $\#T$,

$$\varphi \text{ is a theorem in } T \text{ iff } [\![\varphi]\!] \text{ is a theorem in } \#T, \text{ for any pattern } \varphi.$$

Theorem 4. *The calculus K is faithful for matching logic.*

Proof. TBC. □

Xiaohong: The following paragraphs need revision.

Having a faithful calculus for matching logic has at least the following two benefits. Firstly, any implementation of the calculus is guaranteed to be able to conduct any reasoning in matching logic. Secondly, it allows us to define a matching logic theory T by defining its meta-theory $\#T$ in the calculus. The second point is of great importance if we want a formal language to define matching logic theories. We notice that there are many theories whose definitions involve notations that do not belong to the logic itself. For example, in the (β) axiom

$$\lambda x.e[e'] = e[e'/x] \quad \text{where } e \text{ and } e' \text{ are } \lambda\text{-terms},$$

we use the notation for substitution $\_[\_/\_]$, meta-variables e and e', and their range "λ-terms". None of those can be given a formal semantics in the object-logic, but can be defined in the calculus K.

3 The Kore Language

We have shown K, a calculus for matching logic in which we can specify everything about matching logic and matching logic theories, for example, whether a pattern is well-formed, what sort a patter has, which patterns are deducible, free variables, fresh variables generation, substitution and replacement, alpha-renaming, etc. This meta-theory provides a universe of (meta-representations of) patterns and the proof system of matching logic. On the other hand, it is usually easier to work in the object-level rather than the meta-level. Even if all reasoning in the object-level theory T can be faithfully lifted to and conducted in its meta-theory $\#T$, it does not mean one should always do so.

The Kore language is proposed to define matching logic theories using the calculus K, while it also provides a nice surface syntax (syntactic sugar) to write object-level patterns.

We will show the formal grammar of Kore in Section 3.1, followed by some examples in Section 3.4. After that, we will introduce a transformation from Kore definitions to meta-theories as the formal semantics of Kore in Section 3.3.

3.1 Syntax of Kore

```
// Namespaces for sorts, variables, metavariables,
// symbols, and Kore modules.
Sort           = String
VariableId     = String
MetaVariableId = String
Symbol         = String
ModuleId       = String

Variable       = VariableId:Sort
MetaVariable   = MetaVariableId::Sort

Pattern        = Variable | MetaVariable
               | \and(Pattern, Pattern)
               | \not(Pattern)
               | \exists(Variable, Pattern)
               | Symbol(PatternList)

Sentence       = import ModuleId
               | syntax Sort
               | syntax Sort ::= Symbol(SortList)
               | axiom Pattern
Sentences      = Sentence | Sentences Sentences

Module         = module ModuleId
                   Sentences
                 endmodule
```

3.2 Examples of Kore

The BOOL module.

```
module BOOL
syntax Bool
syntax Bool ::= true | false | notBool(Bool)
| andBool(Bool, Bool) | orBool(Bool, Bool)
axiom  \or(true, false)
axiom  \exists(X:Bool, \equal(X:Bool, true))
axiom  \exists(X:Bool, \equal(X:Bool, notBool(Y:Bool)))
axiom  ... ...
endmodule
```

The NAT module.

```
module NAT
import BOOL
syntax Nat
syntax Nat ::= zero | succ(Nat) | plus(Nat, Nat)
```

```
syntax Bool ::= gt(Nat, Nat) | gte(Nat, Nat)
axiom  \equal(mult(X:Nat, zero), zero)
axiom  \equal(mult(X::Nat, Y::Nat), mult(Y::Nat, X::Nat))
axiom  ... ...
endmodule
```

The LAMBDA module.

```
module LAMBDA
syntax Exp
syntax Exp ::= app(Exp, Exp) | lambda0(Exp, Exp)
syntax K.Bool ::= isLTerm(K.Pattern)

axiom \equals(isLTerm(#variable(X:K.String, #sort("Exp"))), K.true)
axiom \equals(isLTerm(#application(#symbol("app",
(#sort("Exp"), #sort("Exp")),
#sort("Exp")),
(E:K.Pattern, E':K.Pattern))),
K.andBool(isLTerm(E:K.Pattern), isLTerm(E':K.Pattern)))
axiom \equals(isLTerm(#exists(X:K.String, #sort("Exp"),
#application(#symbol("lambda0",
(#sort("Exp"), #sort("Exp")),
#sort("Exp")),
(#variable(X:K.String, #sort("Exp")),
E:K.Pattern))),
isLTerm(E:K.Pattern))
axiom \implies(\equals(K.true,
K.and(isLTerm(E:K.Pattern), isLTerm(E':K.Pattern))),
\equals(K.true,
K.deducible(#equals(...1,
...2))))
endmodule
```

3.3 Semantics of Kore

The semantics of Kore is given as a translation from Kore definitions to meta-theories. The main principle is that every object-level constructors are translated to ground terms (ASTs) while meta-variables are translated to variables in the meta-theories. In other words, the sentence `axiom` φ when φ is an object-pattern is a shorthand of `axiom` $[\![\varphi]\!]$. Since we now allow φ containing meta-variables, the notation $[\![\varphi]\!]$ now means that $mvsc(\varphi) \to (deducible\,(\#\varphi) = true)$, where the lifting function $\#\_$ and the meta-variable sort constraint $mvsc$ are defined in Algorithm 1 and 2, respectively.

Algorithm 1: Lifting Function $\#_{\_}$

Input: An object-pattern φ
Output: The meta-representation of φ

if φ *is* $x{:}s$ **then**
  $\#\mathit{variable}(x, \#\mathit{sort}(s))$
else if φ *is* $x{::}s$ **then**
  $x{:}\mathit{Pattern} \wedge \#\mathit{sort}(s) \in \mathit{getSort}(x{:}\mathit{Pattern})$
else if φ *is* $\varphi_1 \wedge \varphi_2$ **then**
  $\#\mathit{and}(\#\varphi_1, \#\varphi_2)$
else if φ *is* $\neg\varphi_1$ **then**
  $\#\mathit{not}(\#\varphi_1)$
else if φ *is* $\exists x{:}s.\varphi$ **then**
  $\#\mathit{exists}(x, \#\mathit{sort}(s), \#\varphi)$
else if φ *is* $\sigma(\varphi_1, \ldots, \varphi_n)$ **then**
  $\#\mathit{application}(\#\mathit{symbol}(\sigma, \ldots), \#\varphi_1, \ldots, \#\varphi_n)$

Algorithm 2: Meta-Variable Sort Constraint Collection *mvsc*

Input: An object-pattern φ
Output: The meta-variable sort constraint of φ

Collect in set W all meta-variables appearing in φ;
Let $C = \emptyset$;
foreach $x{::}s \in W$ **do**
  $C = C \cup (\#\mathit{sort}(s) \in \mathit{getSort}(x{:}\mathit{Pattern}))$
Return $\bigwedge C$;

4 Object-level and Meta-level

It is an aspect of life in mathematical logics to distinguish the *object-level* and *meta-level* concepts. In matching logic, we put more emphasize and care on metavariables and their range, that is, the set of patterns that they stand for. It turns out that having metavariables that range over all well-formed patterns will lead us to inconsistency theories immediately. As an example, consider the (β) axiom in the matching logic theory LAMBDA of lambda calculus:

$$(\lambda x.e)[e'] = e[e'/x].$$

If we do not put any restriction on the range of metavariables e and e', we have an inconsistency issue as the following reasoning shows:

$$\bot \overset{(\mathrm{N})}{=} (\lambda x.\top)[\bot] \overset{(\beta)}{=} \top[\bot/x] = \top.$$

Therefore, in matching logic, one should explicitly specify the range of metavariables whenever he uses them.

Definition 5 (Restricted metavariables)**.** Let φ be a metavariable of sort $s \in S$. The range of φ is a set of patterns of sort s. We write $\varphi :: R$ if the range of φ is $R \subseteq \mathrm{Pattern}_s$.

Remark 6 (Metavariables in first-order logic)*.* In first-order logic, one often uses metavariables in axiom schemata, but the inconsistency issue does not arise. This is because in first-order logic, we do not need to distinguish metavariables for terms from logic variables, thanks to the next (Substitution) rule:

$$\forall x.\varphi(x) \to \varphi(t).$$

The predicate metavariables are not a problem because there are no object level symbols on top of them.

I don't get the point of predicate metavariables.

Variables and metavariables for variables For any matching logic theory $\mathsf{T} = (S, \Sigma, A)$, it comes for each sort $s \in S$ a countably infinite set V_s of variables. We use $\mathsf{x} : \mathsf{s}, \mathsf{y} : \mathsf{s}, \mathsf{z} : \mathsf{s}, \ldots$ for variables in V_s, and omit their sorts when that is clear from the contexts. Different sorts have disjoint sets of variables, so $\mathrm{Var}_s \cap \mathrm{Var}_{s'} = \emptyset$ if $s \neq s'$.

Proposition 7. *Let A be a set of axioms and $\bar{A} = \forall A$ be the universal quantification closure of A, then for any pattern φ, $A \vdash \varphi$ iff $\bar{A} \vdash \varphi$.*

Remark 8 (Free variables in axioms)*.* The free variables appearing in the axioms of a theory can be regarded as implicitly universal quantified, because a theory and its universal quantification closure are equal.

Example 9.

$$
\begin{aligned}
A_1 &= \{\mathsf{mult}(\mathsf{x}, \mathsf{0}) = \mathsf{0}\} \\
A_2 &= \{\forall \mathsf{x}.\mathsf{mult}(\mathsf{x}, \mathsf{0}) = \mathsf{0}\} \\
A_3 &= \{\forall \mathsf{y}.\mathsf{mult}(\mathsf{y}, \mathsf{0}) = \mathsf{0}\} \\
A_4 &= \{\mathsf{mult}(x, \mathsf{0}) = \mathsf{0}\} \\
A_5 &= \{\forall x.\mathsf{mult}(x, \mathsf{0}) = \mathsf{0}\} \\
A_6 &= \{\forall y.\mathsf{mult}(y, \mathsf{0}) = \mathsf{0}\} \\
A_7 &= \{\mathsf{mult}(\mathsf{x}, \mathsf{0}) = \mathsf{0}, \mathsf{mult}(\mathsf{y}, \mathsf{0}) = \mathsf{0}, \mathsf{mult}(\mathsf{z}, \mathsf{0}) = \mathsf{0}, \dots\} \\
A_8 &= \{\forall \mathsf{x}.\mathsf{mult}(\mathsf{x}, \mathsf{0}) = \mathsf{0}, \forall \mathsf{y}.\mathsf{mult}(\mathsf{y}, \mathsf{0}) = \mathsf{0}, \forall \mathsf{z}.\mathsf{mult}(\mathsf{z}, \mathsf{0}) = \mathsf{0}, \dots\}
\end{aligned}
$$

All the eight theories are equal. Theories A_4, A_5, A_6 are finite representations of theories A_7, A_8, A_8 respectively.

Remark 10. There is no need to have metavariables for variables in the Kore language, because (1) if they are used as bound variables, then replacing them with any (matching logic) variables will result in the same theories, thanks to alpha-renaming; and (2) if they are used as free variables, then it makes no difference to consider the universal quantification closure of them and we get to the case (1).

~~**Given said that, there are cases when metavariables for variables make sense. In those cases we often want our metavariables to range over all variables of all sorts, in order to make our Kore definitions compact.**~~ **No, we do not need metavariables over variables. I was thinking of the definedness symbols. We might want to write only one axiom schema of $\lceil x \rceil$ instead many $\lceil x{:}s \rceil_s^{s'}$'s, but we cannot do that unless we allow polymorphic and overloaded symbols in Kore definitions.**

Patterns and metavariables for patterns It is in practice more common to use metavariables that range over all patterns. One typical example is axiom schemata. For example, $\vdash \varphi \to \varphi$ in which φ is the metavariable that ranges all well-formed patterns.

There has been an argument on whether metavariables for patterns should be sorted or not. Here are some observations. Firstly, since all symbols are decorated and not overloaded, in most cases, the sort of a metavariable for patterns can be inferred from its context. Secondly, the only counterexample against the first point that I can think of is when they appear alone, which is not an interesting case anyway. Thirdly, we do want the least amount of reasoning and inferring in using Kore definitions, so it breaks nothing if not helping things to have metavariables for patterns carrying their sorts.

Example 11.

$$A_1 = \{\mathsf{merge}(\mathsf{h1}, \mathsf{h2}) = \mathsf{merge}(\mathsf{h2}, \mathsf{h1})\}$$
$$A_2 = \{\forall \mathsf{h1} \forall \mathsf{h2}.\mathsf{merge}(\mathsf{h1}, \mathsf{h2}) = \mathsf{merge}(\mathsf{h2}, \mathsf{h1})\}$$
$$A_3 = \{\mathsf{merge}(\varphi, \psi) = \mathsf{merge}(\psi, \varphi)\}$$

All three theories are equal. It is easier to see that fact from a model theoretic point of view, since all theories require that the interpretation of merge is commutative and nothing more. On the other hand, it is not straightforward to obtain that conclusion from a proof theoretic point of view. For example, to deduce $\mathsf{merge}(\mathsf{list}(\mathsf{one}, \mathsf{cons}(\mathsf{two}, \mathsf{epsilon})), \mathsf{top}) = \mathsf{merge}(\mathsf{top}, \mathsf{list}(\mathsf{one}, \mathsf{cons}(\mathsf{two}, \mathsf{epsilon})))$ needs only one step in A_3, but will need a lot more in either A_1 or A_2, because one cannot simply substitute any patterns for universal quantified variables in matching logic.

5 Binders

In matching logic there is a unified representation of binders. We will be using the theory of lambda calculus LAMBDA as an example in this section. Recall that the syntax for untyped lambda calculus is

$$\Lambda ::= V \mid \lambda V.\Lambda \mid \Lambda\ \Lambda$$

where V is a countably infinite set of atomic λ-terms, a.k.a. variables in lambda calculus. The set of all λ-terms, denoted as Λ, is the smallest set satisfying the above grammar.

The matching logic theory LAMBDA has one sort Exp for lambda expressions. It also has in its signature a binary symbol lambda_0 that builds a λ-terms, and a binary symbol app for lambda applications. To mimic the binding behavior of λ in lambda calculus, we define syntactic sugar $\lambda x.e = \exists x.\mathsf{lambda}_0(x, e)$ and $e_1 e_2 = \mathsf{app}(e_1, e_2)$ in theory LAMBDA. Notice that by defining λ as a syntactic sugar using the existential quantifier $\exists x$, we get alpha-renaming for free. The β-reduction is captured by the next axiom:

$$(\lambda x.e)e' = e[e'/x] \quad \text{, where } e \text{ and } e' \text{ are metavariables for } \lambda\text{-terms.}$$

Two important observations are made about the (β) axiom. Firstly, e and e' cannot be replaced by logic variables, because λ-terms in matching logic are (often) not functional patterns. Secondly, metavariables e and e' cannot range over all patterns of sort Exp, but only those which are (syntactic sugar of) λ-terms. Allowing e and e' to range over all patterns of Exp will quickly lead to an inconsistent theory, because of the next contradiction:

$$\bot \overset{(\mathrm{N})}{=} (\lambda x.\top)[\bot] \overset{(\beta)}{=} \top.$$

Therefore, when defining the lambda calculus, we need a way

Theorem 12 (Consistency)**.** *Consider a theory of a binder α, with a sort S and two binary symbols α_0 and $\_\,\_$. Define $\alpha x.e$ as syntactic sugar of $\exists x.\alpha(x,e)$ where x is a variable and e is a pattern. Define α-terms be patterns satisfying the next grammar*

$$T_\alpha ::= V_s \mid \alpha x.T_\alpha \mid T_\alpha T_\alpha.$$

If a theory contains only axioms of the form $e = e'$ where e and e' are α-terms, then the theory is consistent.

Proof. The final model M exists, in which the carrier set is a singleton set, and the two symbols are interpreted as the total function over the singleton set. One can then prove that all α-terms interpret to the total set, so all axioms hold in the final model. □

Corollary 13. *The theory* LAMBDA *is consistent.*

Definition 14 (Common ranges of metavariables)**.**

- Full range Pattern_s;
- Syntactic terms range (variables plus symbols without logic connectives);
- Ground syntactic terms range (symbols only);
- Variable range Var_s (metavariables for variables).

Remark 15. Syntactic terms (and ground syntactic terms) are purely defined syntactically and not equal to terms or functional patterns. When all symbols are functional symbols, the set of syntactic terms equals the set of terms, and both of them are included in the set of all functional patterns.

Remark 16. We need to design a syntax for specifying ranges of metavariables in the Kore language.

Remark 17. We have not proved that matching logic is a conservative extension of untyped lambda calculus, which bothers me a lot. I will remain skeptical about everything we do in this section until we prove that conservative extension result.

The benefit of such a unified theory of binders and binding structures in matching logic is more of theoretical interest. In practice (K backends), one will never want to implement the lambda calculus by desugaring $\lambda x.e$ as $\exists x.\lambda_0(x,\varphi)$ but rather dealing with $\lambda x.\varphi$ directly.

Example 18 (Lambda calculus in Kore)**.**

```
module LAMBDA
  import BOOL
  import META-LEVEL

  syntax Exp
```

```
  syntax Exp ::= app(Exp, Exp)
               | lambda0(Exp, Exp)

  axiom \implies(true = andBool(#isLTerm(#up(E:Exp)),
                                #isLTerm(#up(E':Exp))),
    app(\exists(x:Exp, lambda0(x:Exp, E:Exp)), E':Exp)
      = E:Exp(E':Exp / x:Exp)

  // Q1: what is substitution?
  // Q2: we know #up is not a part of the logic, so what does
  //     it mean?

  syntax Bool ::= #isLTerm(Pattern)
  axiom #isLTerm(#variable(x:Name, s:Sort)) = true
  axiom #isLTerm(#application(
    #symbol(#name("app"), #appendSortList(...), #sort("Exp")))),
    #appendPatternList(#PatternListAsPattern(#P),
                       #PatternListAsPattern(#P')))))
  = andBool(#isLTerm(#P), #isLTerm(#P'))
  ...
endmodule
```

Rewriting logic

6 Contexts

Introduce a binder γ together with its application symbol which we write as $\_[\_]$. Binding variables of the binder γ are often written as $\square$, but in this proposal and hopefully in future work we will use regular variables $x, y, z, \ldots$ instead of $\square$, in order to show that there is nothing special about contexts but simply a theory in matching logic. Patterns of the form $\gamma x.\varphi$ are often called *contexts*, denoted by metavariables $C, C_0, C_1, \ldots$. Patterns of the form $\varphi[\psi]$ are often called *applications*.

Definition 19. The context $\gamma x.x$ is called the identity context, denoted as I. Identity context has the axiom schema $\mathsf{I}[\varphi] = \varphi$ where φ is any pattern.

Example 20. $\mathsf{I}[\mathsf{I}] = \mathsf{I}$.

Definition 21. Let $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is an n-arity symbol. We say σ is *active* on its ith argument ($1 \le i \le n$), if

$$\sigma(\varphi_1, \ldots, C[\varphi_i], \ldots, \varphi_n) = (\gamma x.\sigma(\varphi_1, \ldots, C[x], \ldots, \varphi_n))[\varphi_i],$$

where $\varphi_1, \ldots, \varphi_n$, and C are any patterns. Orienting the equation from the left to the right is often called *heating*, while orienting it from the right to the left is called *cooling*.

Example 22. Assume the next theory of IMP.

$$
\begin{aligned}
A = \{&\mathsf{ite}(C[\varphi], \psi_1, \psi_2) = (\gamma x.\mathsf{ite}(C[x], \psi_1, \psi_2))[\varphi], \\
&\mathsf{while}(C[\varphi], \psi) = (\gamma x.\mathsf{while}(C[x], \psi))[\varphi], \\
&\mathsf{seq}(C[\varphi], \psi) = (\gamma x.\mathsf{seq}(C[x], \psi))[\varphi], \\
&C[\mathsf{ite}(\mathsf{tt}, \psi_1, \psi_2)] \Rightarrow C[\psi_1], \\
&C[\mathsf{ite}(\mathsf{ff}, \psi_1, \psi_2)] \Rightarrow C[\psi_2], \\
&C[\mathsf{while}(\varphi, \psi)] \Rightarrow C[\mathsf{ite}(\varphi, \mathsf{seq}(\psi, \mathsf{while}(\varphi, \psi)), \mathsf{skip})], \\
&C[\mathsf{seq}(\mathsf{skip}, \psi)] \Rightarrow C[\psi]\}.
\end{aligned}
$$

We can simply require that ψ_1, ψ_2, ψ_3, and C are any patterns. That will allow us to do any reasoning that we need, but will that lead to inconsistency?

Example 22(a).

$$
\begin{aligned}
\mathsf{seq}(\mathsf{skip}, \mathsf{skip}) &= \mathsf{I}[\mathsf{seq}(\mathsf{skip}, \mathsf{skip})] \\
&\Rightarrow \mathsf{I}[\mathsf{skip}] \\
&= \mathsf{skip}.
\end{aligned}
$$

Example 22(b).

$$
\begin{aligned}
\mathsf{seq}(\mathsf{ite}(\mathsf{tt}, \psi_1, \psi_2), \psi_3) &= \mathsf{seq}(\mathsf{I}[\mathsf{ite}(\mathsf{tt}, \psi_1, \psi_2)], \psi_3) \\
&= (\gamma x.\mathsf{seq}(\mathsf{I}[x], \psi_3))[\mathsf{ite}(\mathsf{tt}, \psi_1, \psi_2)] \\
&\Rightarrow (\gamma x.\mathsf{seq}(\mathsf{I}[x], \psi_3))[\psi_1] \\
&= \mathsf{seq}(\mathsf{I}[\psi_1], \psi_3) \\
&= \mathsf{seq}(\psi_1, \psi_3).
\end{aligned}
$$

Example 23. Consider the following theory written in the Kore language:

```
module IMP
  import ...
  syntax AExp
  syntax AExp ::= plusAExp(AExp, AExp)
  syntax AExp ::= minusAExp(AExp, AExp)
  syntax AExp ::= AExpAsNat(Nat)
  syntax BExp
  syntax BExp ::= geBExp(AExp, AExp)
  syntax BExp ::= BExpAsBool(Bool)
  syntax Pgm
  syntax Pgm  ::= skip()
  syntax Pgm  ::= seq(Pgm Pgm)
  syntax Pgm  ::=
  syntax Heap
```

```
  syntax Cfg

endmodule
```

Example 24. Following the above example, extend A with the next axioms:

$$\begin{aligned}&\{C[x][\mathsf{mapsto}(x,v)] \Rightarrow C[v][\mathsf{mapsto}(x,v)],\\ &\ C[\mathsf{asgn}(x,v)][\mathsf{mapsto}(x,v')] \Rightarrow C[\mathsf{skip}][\mathsf{mapsto}(x,v)],\\ &\ C[\mathsf{asgn}(x,v)][\varphi] \Rightarrow C[\mathsf{skip}][\mathsf{merge}(\varphi,\mathsf{mapsto}(x,v))]\}\end{aligned}$$

The above example is meant to show the loopup rule, but it does not work because the third axiom is incorrect. Instead of simply writing φ, we should say that φ does not assign any value to x. One solution (that is used in the current K backend) is to introduce a strategy language and to extend theories with strategies.

Example 25. Suppose f and g are binary symbols who are active on their first argument. Suppose a, b are constants, and x is a variable. Let $\square_1$ and $\square_2$ be two hole variables. Define two contexts $C_1 = \gamma\square_1.f(\square_1, a)$ and $C_2 = \gamma\square_2.g(\square_2, b)$.

Because f is active on the first argument,

$$\begin{aligned}C_1[\varphi] &= (\gamma\square_1.f(\square_1, a))[\varphi]\\ &= (\gamma\square_1.f(\mathsf{I}[\square_1], a))[\varphi]\\ &= f(\mathsf{I}[\varphi], a)\\ &= f(\varphi, a),\ \text{for any pattern } \varphi.\end{aligned}$$

And for the same reason, $C_2[\varphi] = g(\varphi, b)$. Then we have

$$\begin{aligned}C_1[C_2[x]] &= C_1[f(x,a)]\\ &= g(f(x,a),b).\end{aligned}$$

On the other hand,

$$\begin{aligned}g(f(x,a),b) &= g(C_1[x],b)\\ &= (\gamma\square.g(C_1[\square],b))[x]\\ &= (\gamma\square.g(f(\square,a),b))[x].\end{aligned}$$

Therefore, the context $\gamma\square.g(f(\square,a),b))$ is often called the *composition* of C_1 and C_2, denoted as $C_1 \circ C_2$.

Example 26. Suppose f is a binary symbol with all its two arguments active. Suppose C_1 and C_2 are two contexts and a, b are constants. Then easily we get

$$\begin{aligned}f(C_1[a], C_2[b]) &= (\gamma\square_2.f(C_1[a], C_2[\square_2]))[b]\\ &= (\gamma\square_2.((\gamma\square_1.f(C_1[\square_1], C_2[\square_2]))[a]))[b].\end{aligned}$$

What happens above is similar to *curring* a function that takes two arguments. It says that there exists a context C_a, related with C_1, C_2, f and a of course, such that $C_a[b]$ returns $f(C_1[a], C_2[b])$. The context C_a has a binding hole $\square_2$, and a body that itself is another context C'_a applied to a. In other words, there exists C_a and C'_a such that

- $f(C_1[a], C_2[b]) = C_a[b]$,
- $C_a = \gamma\square_2.(C'_a[a])$,
- $C'_a = \gamma\square_1.f(C_1[\square_1], C_2[\square_2])$.

A natural question is whether there is a context C such that $C[a][b] = f(C_1[a], C_2[b])$.

Proposition 27. $C_1[C_2[\varphi]] = C[\varphi]$, *where* $C = \gamma\square.C_1[C_2[\square]]$.

6.0.1 Normal forms

In this section, we consider *decomposition* of patterns. A decomposition of a pattern P is a pair $\langle C, R\rangle$ such that $C[R] = P$. Let us now consider patterns that do not have logical connectives.

Fixed points

7 Appendix: The First Kore Language

The next grammar is the firstly-proposed Kore language at here.

```
Definition = Attributes
Set{Module}

Module = module ModuleName
Set{Sentence}
endmodule
Attributes

Sentence =  import ModuleName Attributes
| syntax Sort Attributes                        // sort declarations
| syntax Sort ::= Symbol(List{Sort}) Attributes    // symbol declarations
| rule Pattern Attributes
| axiom Pattern Attributes

Attributes = [ List{Pattern} ]

Pattern =  Variable
| Symbol(List{Pattern})                          // symbol applications
| Symbol(Value)                                  // domain values
| \top()
| \bottom()
| \and(Pattern, Pattern)
```

```
| \or(Pattern, Pattern)
| \not(Pattern)
| \implies(Pattern, Pattern)
| \exists(Variable, Pattern)
| \forall(Variable, Pattern)
| \next(Pattern)
| \rewrite(Pattern, Pattern)
| \equals(Pattern, Pattern)

Variable =  Name:Sort                                         // variables

ModuleName = RegEx1
Sort       = RegEx2
Name       = RegEx2
Symbol     = RegEx2
Value      = RegEx3

RegEx1 == [A-Z][A-Z0-9-]*
RegEx2 == [a-zA-Z0-9.@#$%^_-]+ | ‘ [^‘]* ‘
RegEx3 == <Strings>   // Java-style string literals, enclosed in quotes
```

In the grammar above, `List{X}` is a special non-terminal corresponding to possibly empty comma-separated lists of `X` words (trivial to define in any syntax formalism). `Set{X}`, on the other hand, is a special non-terminal corresponding to possibly empty space-separated sets of X words. Syntactically, there is no difference between the two (except for the separator), but Kore tools may choose to implement them differently.

7.1 Builtin theories

```
module BOOL
syntax Bool
syntax Bool ::= true | false | notBool(Bool)
| andBool(Bool, Bool) | orBool(Bool, Bool)

// axioms for functional symbols
axiom \exists(T:Bool, \equals(T:Bool, true))
axiom \exists(T:Bool, \equals(T:Bool, false))
axiom \exists(T:Bool, \equals(T:Bool, \notBool(X:Bool)))
axiom \exists(T:Bool, \equals(T:Bool, andBool(X:Bool, Y:Bool)))
axiom \exists(T:Bool, \equals(T:Bool, orBool(X:Bool, Y:Bool)))

// axioms for commutativity
axiom \equals(andBool(X:Bool, Y:Bool), andBool(Y:Bool, X:Bool))
axiom \equals(orBool(X:Bool, Y:Bool), orBool(Y:Bool, X:Bool))

// the no-junk axiom for constructors
```

```
axiom \or(true, false)

axiom \equals(notBool(true), false)
axiom \equals(notBool(false), true)
axiom \equals(andBool(true, T:Bool), T:Bool)
axiom \equals(andBool(false, T:Bool), false)
axiom \equals(orBool(true, T:Bool), true)
axiom \equals(orBool(false, T:Bool), T:Bool)
endmodule

module META-LEVEL
syntax
endmodule

module LAMBDA
syntax Exp
syntax Exp ::= lambda0(Exp, Exp) | app(Exp, Exp)

endmodule
```