The Semantics of K

Formal Systems Laboratory
University of Illinois

August 11, 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.

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 [?]. Let *Var* be a countable set of *variables*. Assume a matching logic *signature* (S, Σ). 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.

Not sure why you prefer to work with only one set of variables, instead of a set Var_s for each sort s.

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

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 $\_*\_(\_+\_(x{:}Nat, y{:}Nat), z{:}Nat)$.

Remark 1. The above is a formal grammar, whose semantics[1] is well studied and crystal clear. The grammar defines for each sort s exactly one set of well-formed patterns of sort s. We are defining a *set*, that is to say, following the

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.

I think we do not need all this discussion here; we want to keep the document clean and short, to serve as an authoritative reference

[1] Semantics as a formal grammar, not the matching logic semantics.

above grammar, one is able to (1) distinguish well-formed patterns from ill-formed ones; and (2) decide whether two well-formed patterns are the same pattern or not. The next remark provides an example of (2).

Remark 2. Assume $x, y \in N$ and $s \in S$, pattern $x{:}s \wedge y{:}s$ is a well-formed pattern of sort s, by definition. It is also *distinct* from pattern $y{:}s \wedge x{:}s$, by definition. The fact that (after introducing equalities as syntactic sugar in the logic) one can use some proof systems of matching logic and establish

$$\vdash x{:}s \wedge y{:}s = y{:}s \wedge x{:}s,$$

or use the semantics of matching logic and establish

$$\vDash x{:}s \wedge y{:}s = y{:}s \wedge x{:}s,$$

has *nothing* to do with the formal grammar itself and does not change the fact that $x{:}s \wedge y{:}s$ and $y{:}s \wedge x{:}s$ are two distinct patterns in matching logic.

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

$\varphi[\varphi_1/x] \wedge (\varphi_1 = \varphi_2) \to \varphi[\varphi_2/x]$	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
$\varphi_1 \mathtt{+} \varphi_2 = \varphi_1 +_{Nat} \varphi_2$	where φ, φ' are *ground* syntactic patterns of sort *Nat*, that is, patterns built only with symbols `zero` and `succ`
$(\varphi_1 \to \varphi_2) \to (\varphi[\varphi_1/x] \to \varphi[\varphi_2/x])$	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.

Let's introduce this when needed. Also, "equal" is not a good word. "Two theories are *equal* if they have the same set of sorts and symbols, and they deduce the same set of theorems."

2 A Calculus of Matching Logic

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

> 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 the many-sorted equational logic as the *meta-logic*[2]. The calculus of matching logic, denoted as $M = (S_M, \Sigma_M, E_M)$, is a many-sorted equational logic where S_M is a set of sorts, Σ_M is a set of functional and relational symbols, and E_M is a set of equations, defined as the next Maude theory:

```
fth M is
  protecting STRING .

  sort Sort . op #sort : String -> Sort .
  sort SortList .          ---- comma-separated lists

  sort Symbol .
  op #symbol : String      ---- unique identity / name / reference
               SortList    ---- argument sorts
               Sort        ---- result sort
               -> Symbol .

  sorts K KList .          ---- Grigore proposed to name it K
                           ---- instead of Pattern

  op #variable : String    ---- unique identity / name / reference
                 Sort -> K .
  op #and : K K -> K .
  op #not : K -> K .
  op #exists : String      ---- id of binding variable
               Sort        ---- sort of binding variable
               K -> K .
  op #application : Symbol KList -> K .
  op #value : String       ---- encoding of domain values
              Sort         ---- sort of values
              -> K .       ---- not sure about #value

  op well-formed : K -> Bool .
  op getSort : K -> [Sort] .
  op isSort : K Sort -> Bool .

  cmb getSort(K:K) : Sort if well-formed(K:K) [nonexe] .
  eq isSort(K:K, S:Sort) = (getSort(K:K) = S:Sort) [nonexe] .

  op getFreeVariables : K
                        -> ... .   ---- a collection of variables

  ---- fine-grained-controlled substitution
  op replace : K           ---- the main pattern
               K           ---- the "replace" pattern
               K           ---- the "find" pattern
               ...         ---- some controlling arguments
```

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

```
              -> K .

    op freshName : StringList      ---- a collection of used names.
                   -> String .

    ---- the following is about the proof system of matching logic.
    ---- Reference: [L. Paulson] The Foundation of a Generic Theorem Prover

    sorts InferenceRule      ---- a function from theorems to theorems
          Tactic             ---- the "reverse" of an inference rule
          Validation         ---- the "certificate" returned by a tactic
          Tactical .         ---- control structures applied on tactics

    op deducible : K -> Bool .

    ---- not finished
  endth
```

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

Suppose L is a syntactic matching logic theory whose axiom set is A. Let $\#L$ be the meta-level theory of L obtained by adding

$$[\![A]\!] = \{\mathit{deducible}\,([\![\varphi]\!]) \mid \varphi \in A\}$$

as axioms to the meta-logic theory M, where the double bracket $[\![\_]\!]$ is a function that maps object-patterns to their meta-representations in M. The following definition is inspired by Paulson's paper (Definition 1).

Definition 3. Let L be a matching logic theory and $\#L$ is its meta-logic theory. Let $\varphi_1, \ldots, \varphi_n$ and ψ be patterns of L. Then say

- $\#L$ is *sound* for L, if for every $\#L$-proof of $\mathit{deducible}\,([\![\psi]\!])$ from $\mathit{deducible}\,([\![\varphi_1]\!])$, $\ldots$, $\mathit{deducible}\,([\![\varphi_n]\!])$, there is an L-proof of ψ from $\varphi_1, \ldots, \varphi_n$.
- $\#L$ is *complete* for L, if for every L-proof of ψ from $\varphi_1, \ldots, \varphi_n$, there is a $\#L$-proof of $\mathit{deducible}\,([\![\psi]\!])$ from $\mathit{deducible}\,([\![\varphi_1]\!]), \ldots, \mathit{deducible}\,([\![\varphi_n]\!])$.
- $\#L$ is *faithful* for L if it is both sound and complete.

As a result of $\#L$ being faithful for L, for any pattern φ, φ is deducible in L iff $\mathit{deducible}(\varphi)$ is deducible in $\#L$.

For example, the next Maude functional theory defines the meta-logic of lambda calculus in matching logic:

```
fth M-LAMBDA is
  extending M .
```

```
  ---- the following syntactic sugar is just for readability.

  ops app lambda0 : -> Symbol .
  eq app = #symbol("app", ("Exp", "Exp"), "Exp") .
  eq lambda0 = #symbol("lambda0", ("Exp", "Exp"), "Exp") .

  op lambda_._ : String K -> K .
  eq lambda X:VariableId . E:K
   = #exists(X:VariableId, "Exp",
     #application(lambda0, (#variable(X:VariableId, "Exp"), E:K)))) .
  op _[_] : K K -> K .
  eq E1:K[E2:K]
   = #application(app, (E1:K, E2:K))) .

  ---- side conditions checker

  op isLTerm : K -> Bool .

  eq isLTerm(#variable(X:VariableId, "Exp")) = true .

  eq isLTerm(E1:K[E2:K])
   = isLTerm(E1:K) and isLTerm(E2:K) .

  eq isLTerm(lambda(X:VariableId, E:K))
   = isLTerm(E:K) .

  ---- the (Beta) axiom

  cmb #equal((lambda X:VariableId . E1:K)[E2:K],
             replace(...)) : Theorem
   if isLTerm(E1:K) and isLTerm(E2:K) .
endth
```

One can see how complex it is to write meta-logic theories, but it is an aspect of life. In the next section, we will introduce the Kore language that lets one define theories in the object-level.

3 The Kore Language

We have shown a meta-logic theory M in which we can specify everything about 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-logic theory provides a universe of (meta-representations of) patterns, the proof system of matching logic, the entailment relation, etc. together with all kinds of operations and functions. On the other hand, it is easier to work in the object-level rather than the meta-level. Even if all reasoning in the object-logic L can be faithfully lifted to the meta-logic theory $\#L$, it does not mean one should always do so.

The Kore language is proposed to help writing a matching logic theory L mainly at the object-level, and only go to the meta-level if it is explicitly required. The outcomes are called Kore definitions, whose semantics is given by defining a transformation that maps a Kore definition `module.kore` to a meta-logic theory $\#L$. In this case we say that `module.kore` defines the matching logic theory L.

The following starts to become shaky.

We have defined an equational theory M as the calculus of matching logic, which means all reasoning about matching logic theories can be faithfully lifted to M (we haven't proved faithfulness, though.) In theory, by having (a faithful) M with us, we are good to go. We can define whatever things we like and we want about object-level theories in the meta-logic M, and we are not going to propose a language syntax to write equational logic theories anyway.

What happens now is that in practice, we want to have a language, called Kore, that allows us to work at the object-level (for convenience), while still allows us to touch the meta-logic theory M whenever we need, so that we are in theory able to define everything.

Therefore, the Kore language is a mixture of object-level and meta-level. Remember that the object-level is matching logic, while meta-level is equational logic, so we are proposing a language in which users can write some matching logic things intertwined with equational logic things. The two levels things can appear in the same line, such as:

```
   isLterm(app(E::Exp, E'::Exp))
= isLTerm(E::Exp) and isLTerm(E'::Exp)
```

in which `isLTerm` is a predicate symbol at meta-level, `app` is a matching logic symbol, `E::Exp`, `E'::Exp` are meta-variables of matching logic patterns (i.e., variables of meta-representations of patterns), `"="` is the equality at meta-level, and `and` is at meta-level.

We propose the next Kore syntax.

```
// Namespaces for sorts, variables, metavariables,
// symbols, and Kore modules.
SortId         = String
VariableId     = String
MetaVariableId = MaudeId
SymbolId       = String
ModuleId       = String

Sort           = SortId

Variable       = VariableId:SortId
MetaVariable   = MetaVariableId::SortId

Pattern        = Variable | MetaVariable
```

```
               | \and(Pattern, Pattern)
               | \not(Pattern)
               | \exists(Variable, Pattern)
               | SymbolId(List{Pattern})

Signature      = syntax SortId
               | syntax SortId ::= SymbolId(List{SortId})
               | Signature Signature

Axioms         = axiom Pattern
               | Axioms Axioms

Module         = module ModuleId
                   Signature
                   Axioms
                 endmodule
```

3.1 Example Kore Definitions

The BOOL module.

```
module BOOL
  syntax Bool
  syntax Bool ::= trueBool | falseBool | notBool(Bool)
               | andBool(Bool, Bool) | orBool(Bool, Bool)
  axiom  \or(trueBool, falseBool)
  axiom  \exists(X:Bool, \equal(X:Bool, trueBool))
  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)
              | mult(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  ... ...
  // hook to a model (not finished)
  hook-sort Nat "external-model Naturals"
  hook-symbol zero \value("0", Nat)
  hook-symbol succ \value("succ", Nat)
  ... ...
endmodule
```

The LAMBDA module.

```
module LAMBDA
  import M       ---- import the meta-logic theory M
                 ---- we can rename it to K ...
  syntax Exp
  syntax Exp ::= app(Exp, Exp)
               | lambda0(Exp, Exp)
  syntax M.Bool ::= isLTerm(K)      ---- I think the Bool here is the
                                    ---- one in the meta-logic M, not
                                    ---- not one in the Kore module
                                    ---- BOOL
  axiom  \equals(app(\exists(X:Exp, lambda0(X:Exp, E::Exp)), E'::Exp),
                \replace(E::Exp, E'::Exp, X:Exp))
  requires \isLTerm(E::Exp) M.and \isLTerm(E'::Exp) ---- requires is a sugar

  axiom isLTerm(X:Exp) = M.true
  axiom isLTerm(app(E::Exp, E'::Exp)) = isLTerm(E::Exp) M.and isLTerm(E'::Exp)
  ...

endmodule
```

3.2 Semantics of Kore

We give Kore definitions semantics by showing how to translate (desugar) them to meta-logic theories.

Desugaring Kore definitions to meta-logic theories. This has top priority now. Once we have that transformation, we can claim a formal semantics of Kore.

The following rules transform Kore objects to their meta-representations. The principle is that every object-level things in Kore become ground terms in the meta-logic. Every meta-level things in Kore, for example meta-variables, become variables in the meta-logic.

The next transformation needs to be nailed down a bit.

```
#up(X:Nat) => #variable("X", "Nat")
#up(X::Nat) => X:Pattern "with" isSort(X:Pattern, "Nat")
#up(\and(P, Q)) => #and(#up(P), #up(Q))
#up(\not(P)) => #not(#up(P))
#up(\exists(X:Nat, P)) => #exists("X", "Nat", #up(P))

#up(axiom P) => cmb #up(P) : Theorem if isSort(...) /\ ... .
```

3.3 Lambda Calculus

```
module LAMBDA
  syntax Exp
```

```
  syntax Exp  ::= lambda0(Exp, Exp)
                | app(Exp, Exp)
  syntax Bool ::= isLTerm(Pattern)
  axiom isLTerm(X:Exp) = true
  axiom isLTerm(\exists(X:Exp, lambda0(X:Exp, E:Exp)))
      = isLTerm(E:Exp)
  axiom isLTerm(app(E:Exp, E':Exp))
      = isLTerm(E:Exp) andBool isLTerm(E':Exp)
  axiom app(\exists(X:Exp, lambda0(X:Exp, E::Exp)), E'::Exp)
      = E::Exp[E'::Exp / X:Exp]
  requires isLTerm(E::Exp) andBool isLTerm(E'::Exp)
endmodule
```

Many discussions in the next section (Sec.4 Object-level and Meta-level) should be moved to Section 3 as examples. Sec.5 Binders and Sec.6 Contexts should also move to a subsection of Sec.3 as applications and examples.

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 4 (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 5 (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 6. *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 7 (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 8.

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

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 9*.* 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 10.

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

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 11 (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 12. *The theory* LAMBDA *is consistent.*

Definition 13 (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 14. 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 15. We need to design a syntax for specifying ranges of metavariables in the Kore language.

Remark 16. 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 17 (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 18. 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 19. $\mathsf{I}[\mathsf{I}] = \mathsf{I}$.

Definition 20. Let $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is an n-arity symbol. We say σ is *active* on its ith argument ($1 \leq i \leq n$), 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 21. 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 21(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 21(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 22. 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 23. 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 24. 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,

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

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

Example 25. 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

$$\begin{aligned} f(C_1[a], C_2[b]) &= (\gamma\Box_2.f(C_1[a], C_2[\Box_2]))[b] \\ &= (\gamma\Box_2.((\gamma\Box_1.f(C_1[\Box_1], C_2[\Box_2]))[a]))[b]. \end{aligned}$$

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 $\Box_2$, 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

- $f(C_1[a], C_2[b]) = C_a[b]$,
- $C_a = \gamma\Box_2.(C'_a[a])$,
- $C'_a = \gamma\Box_1.f(C_1[\Box_1], C_2[\Box_2])$.

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

Proposition 26. $C_1[C_2[\varphi]] = C[\varphi]$, *where* $C = \gamma\Box.C_1[C_2[\Box]]$.

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
```