The Semantics of K

Formal Systems Laboratory
University of Illinois

August 5, 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.

1 Preliminary

Definition 1 (Matching logic theory)**.** A matching logic theory (S, Σ, A) is a triple that contains a nonempty finite set of sorts, a finite or countably infinite set of symbols, and a recursive set of axioms. Two theories are *equal* if they have the same set of sorts and symbols, and they deduce the same set of theorems.

Example 2. We use serif fonts to denote matching logic theories. Some of the commonly used ones are the theory (theories) of definedness DEF, the theory of Presburger arithmetic PA, the theory of sequences of natural numbers SEQ, the theory of memory heaps HEAP, the theory of IMP programs IMP, the theory (theories) of fixed-points FIX, and the theory (theories) of contexts CTXT.

Definition 3 (The Kore Language)**.** The Kore language is a language to write matching logic theories. The outcomes are called Kore definitions. Kore definitions are mainly served as the interface between a K frontend and a K backend, but a human should be able to read and write Kore definitions of simple theories, too. The Kore language is designed in a way that:

- Every Kore definition defines exactly one matching logic theory;
- Every matching logic theory can be defined as a Kore definition;
- There is no parsing ambiguity.
- The least amount of inferring is needed;
- Symbols are decorated with their argument sorts and result sort;
- There is no polymorphic or overloaded symbols, so every symbol has a unique name (reference).
- And more ...

Definition 4 (Frontend)**.** A K frontend is an artifact that generates Kore definitions.

Definition 5 (Backend)**.** A K backend is an artifact that consumes a Kore definition of a theory T and does some work. Whatever it does can and should be algorithmically reduced to the task of proving $\mathsf{T} \vdash \varphi$ where φ "encodes" that work. A K backend should justify its results by generating formal proofs that can be proof-checked by the oracle matching logic proof checker. Examples of K backends include concrete execution engines, symbolic executions engines, matching logic provers, verification tools, etc.

2 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, for example, 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 6 (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 7 (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 8. *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 9 (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 10.

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

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 11. 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 12.

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

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.

This is a nice example that shows the power of metavariables and how they greatly shorten formal proofs, but this is not a good example to convince people that we need metavariables for patterns in order to embrace more expressiveness. There is an example in the theory LAMBDA that clearly shows metavariables give us more expressiveness power, and we will cover that example in later sections.

2.1 The module META − LEVEL

The theory of meta-level in matching logic provides a universe of metarepresentation (abstract syntax trees) of object-level patterns. The theory of meta-level helps us specify the range of metavariables and much more. One important observation is that the connection between object-level modules and the meta-level module cannot be defined in the logic itself. More specifically, the operator #up that takes object-level patterns to their metarepresentation in the meta-level module cannot be defined in the logic no matter how. Consider the next simple example where one defines a symbol isGround that checks whether a pattern is

a ground pattern or not, using the meta-level module:

$$
\begin{aligned}
&\mathsf{isGround}(\#\mathsf{up}[X]) = \mathsf{false}\\
&\mathsf{isGround}(\#\mathsf{up}[\sigma]) = \mathsf{true}\\
&\mathsf{isGround}(\#\mathsf{up}[\sigma(\varphi_1, \ldots, \varphi_n)]) =\\
&\quad \mathsf{andBool}(\mathsf{isGround}(\#\mathsf{up}[\varphi_1]), \ldots, \mathsf{isGround}(\#\mathsf{up}[\varphi_n]))\\
&\mathsf{isGround}(\#\mathsf{up}[\varphi \wedge \psi]) =\\
&\quad \mathsf{andBool}(\mathsf{isGround}(\#\mathsf{up}[\varphi]), \mathsf{isGround}(\#\mathsf{up}[\psi]))\\
&\ldots
\end{aligned}
$$

Unfortunately, the first two axioms lead us to inconsistency, as shown in the following:

$$
\begin{aligned}
&\mathsf{isGround}(\#\mathsf{up}[X - X]) = \mathsf{false}\\
&\mathsf{isGround}(\#\mathsf{up}[\mathsf{zero}]) = \mathsf{true}
\end{aligned}
$$

The only way to prevent such inconsistency issue is to prevent equations propagating through $\#\mathsf{up}$, more specifically, to forbid the next inference rule:

$$\text{From } \varphi = \psi \text{ deduce } \#\mathsf{up}[\varphi] = \#\mathsf{up}[\psi],$$

which is a strong evidence that $\#\mathsf{up}$ is not part of the logic.

We provide the Kore definition of the $\mathsf{META-LEVEL}$ module.

```
module META-LEVEL
  import STRING

  syntax Sort
  syntax SortList
  syntax Symbol
  syntax Name

  syntax Sort ::= #sort(String)
  syntax Name ::= #name(String)
  syntax Symbol ::= #symbol(Name, SortList, Sort)
  syntax SortList ::= #nilSortList()
                    | SortListAsSort(Sort)
                    | #appendSortList(SortList, SortList)
  syntax Pattern
  syntax Pattern ::= #variable(Name, Sort)
                   | #and(Pattern, Pattern)
                   | #not(Pattern)
                   | #exists(Name, Sort, Pattern)
                   | #application(Symbol, PatternList)

  syntax PatternList
```

```
  syntax PatternList ::= #nil()
                       | #PatternListAsPattern(Pattern)
                       | #append(PatternList, PatternList)

  syntax SortSet SymbolSet PatternSet // defined in the usual way

  syntax Module
  syntax Module ::= #module(SortSet, SymbolSet, PatternSet)

  syntax Bool ::= #wellFormed(Pattern)
  syntax Sort ::= #getSort(Pattern)
  // more to go

endmodule
```

Using the $\mathsf{META-LEVEL}$ module, we can easily define the ranges of metavariables using *side conditions*. The next section illustrates that.

3 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 13 (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 14. *The theory* LAMBDA *is consistent.*

Definition 15 (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 16. 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 17. We need to design a syntax for specifying ranges of metavariables in the Kore language.

Remark 18. 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 19 (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

4 Contexts

Introduce a binder γ together with its application symbol which we write as $\_[\_]$. Binding variables of the binder γ are often written as $\Box$, but in this proposal and hopefully in future work we will use regular variables $x, y, z, \ldots$ instead of $\Box$, 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 20. 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 21. $\mathsf{I}[\mathsf{I}] = \mathsf{I}$.

Definition 22. 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 23. 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 23(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 23(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 24. 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 25. 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 26. 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 $\Box_1$ and $\Box_2$ be two hole variables. Define two contexts $C_1 = \gamma\Box_1.f(\Box_1, a)$ and $C_2 = \gamma\Box_2.g(\Box_2, b)$.

Because f is active on the first argument,

$$\begin{aligned}C_1[\varphi] &= (\gamma\Box_1.f(\Box_1,a))[\varphi]\\ &= (\gamma\Box_1.f(\mathsf{I}[\Box_1],a))[\varphi]\\ &= f(\mathsf{I}[\varphi],a)\\ &= f(\varphi,a),\ \text{for any pattern } \varphi.\end{aligned}$$

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 27. 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 28. $C_1[C_2[\varphi]] = C[\varphi]$, *where* $C = \gamma\Box.C_1[C_2[\Box]]$.

4.0.1 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

5 The 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.

5.1 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
```