The Semantics of K

Formal Systems Laboratory
University of Illinois

August 10, 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.

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

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

I think we also need to talk about: other logical connectives as derived, free variables, capture-free substitution; maybe more

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 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, axioms may be presented as *axiom 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, patterns formed only with variables and symbols

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 The Theory of Matching Logic

Remark 3. We call matching logic the *object-logic*, in comparison to the meta-logic, in which we define a theory of matching logic and its proof system.

Some discussions. The main task of this proposal is to find a nice way to specify (and probably reason with) matching logic theories. We spent lots of time considering the design of the Kore language, with which we specify matching logic theories, and we decided to go for meta-logic. This paragraph is a summary that serves as a justification of our choice.

Suppose we allow users to write infinite axioms to specify theories. In that case, there is no need to go for a meta-logic, since one can always enumerate all axioms in Kore definitions, as long as they are recursively enumerable (r.e.). Theories who do not have a r.e. set of axioms are not considered as of interest of practice.

However, we cannot afford our definitions to run forever. We look for a finite representation of them. This is when the concept of meta-variables comes in and plays a role. And soon one realizes that we do not only need meta-variables, but also some notations in the meta-level, too, serving as methods that help us finitely represent the infinite sets of axioms we want to specify. One typical such notation is the substitution $\varphi[\psi/x]$, where $\_[\_/\_]$ is the mathematical notation, in the meta-level, for substitution, which is not part of the object-logic. Other examples include side-conditions on meta-variables and context.

We look for a generic way that helps us represent infinite axioms in a finite way. And there is a natural one. Since axioms are r.e. sets, by definition there are semi-decision procedures that define them, and semi-decision procedures have finite representation. Therefore, instead of enumerating all the axioms, one simply write down the corresponding semi-decision procedure.

In this section, we will propose a meta-logic that serves as *a logic for matching logic and its proof system.* We believe such as meta-logic will result in a direct implementation of Kore and matching logic provers.

The meta-logic needs not to be the matching logic. In fact, it can simply be a rewriting logic or even equational logic, and we can write it down as a Maude theory. We only use Maude's syntax to specify theories in the meta-logic, so those Maude modules (theories) need no to be executable. In fact, we assume every equations are labeled with the attribute `nonexe`.

Definition 4 (Theory of Matching Logic)**.** The theory of matching logic, denoted as U, is a theory of meta-logic together with a lifting operator $\langle \cdot \rangle$, such that for any matching logic theory T and pattern φ,

$$T \vdash \varphi \quad \text{iff} \quad U \vdash \langle T \vdash \varphi \rangle.$$

The theory U is also called the *universal theory.*

Remark 5. Think of the universal theory U as *the theory of ASTs of patterns.*

Remark 6. The universal theory U is like the universal Turing machine.

```
fth META-LEVEL
  protecting STRING .
  sorts Symbol Pattern PatternList StringList.

  subsort String < StringList .
  op nilStringList : -> StringList .
  op _,_ : StringList StringList -> StringList [assoc] .

  subsort Pattern < PatternList .
  op nilPatternList : -> PatternList .
  op _,_ : PatternList PatternList -> PatternList [assoc] .

  op #symbol : String StringList String -> Symbol .

  op #variable : String String -> Pattern .
```

```
    op #and : Pattern Pattern -> Pattern .
    op #not : Pattern -> Pattern .
    op #exists : String String Pattern -> Pattern .
    op #application : Symbol PatternList -> Pattern .
    ---- more things to go ----
    ---- well-formed, getSort, isSort, freshvar, substitution, etc. ----

    sort Theorem . ---- patterns that can be deducted
    subsort Theorem < Pattern .

    ---- just an example of logic axioms
    mb #not(#and(P:Pattern, #not(P:Pattern))) : Theorem .
  endth
```

The next example is (the meta-representation of) the theory of lambda calculus. Again, equations are assumed to have the attribute `nonexe`.

```
fth LAMBDA
  including META-LEVEL .

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

  ---- side conditions checker

  op isLTerm : Pattern -> Bool .

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

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

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

```
  ---- the (Beta) axiom
  ---- we haven't defined #equal in the meta-logic yet
  ---- nor #substitute
  cmb #equal((lambda X:VariableId . E1:Pattern)[E2:Pattern],
             #substitute(...)) : Theorem
   if isLTerm(E1:Pattern) and isLTerm(E2:Pattern) .
endth
```

3 The Kore Language

We have a meta-logic (the equational logic) to specifies everything about matching logic theories, including whether a pattern is well-formed, what sort a patter has, which patterns are deducible, free variables, fresh variables generation, substitution, alpha-renaming, etc. This theory of meta-level we have just defined provides you a universe of (meta-representations of) patterns, together with all kinds of operations and functions that help you do whatever you want, so one has full control in the meta-logic.

On the other hand, working in the object-level is always more intuitive and friendly than working in the meta-level. The meta-level exists so that we know we are able to do everything, but it does not necessarily mean one should always go to meta-level and work there. We propose the Kore language, which provides users a friendly way to write theories in the object-level, which will eventually be desugared to the meta-logic.

Definition 7 (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 it can be desugared to a theory in the meta-logic in a deterministic way.

Definition 8 (Frontend)**.** A K frontend is an artifact that generates Kore definitions.

Definition 9 (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.

We proposed the next Kore language syntax.

```
// Namespaces for sorts, variables, metavariables,
// symbols, and Kore modules.
```

```
SortId         = ...
VariableId     = ...
MetaVariableId = ...
SymbolId       = ...
ModuleId       = ...

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 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(...) /\ ... .
```

3.2 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 10 (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 11 (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 12. *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 13 (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 14.

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

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 15. 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 16.

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

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:

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

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

Theorem 17 (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 18. *The theory* LAMBDA *is consistent.*

Definition 19 (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 20. 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 21. We need to design a syntax for specifying ranges of metavariables in the Kore language.

Remark 22. 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 23 (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 24. 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 25. $\mathsf{I}[\mathsf{I}] = \mathsf{I}$.

Definition 26. 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 27. 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 27(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 27(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 28. 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 29. 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 30. 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 31. 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 32. $C_1[C_2[\varphi]] = C[\varphi]$, *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
```