The Semantics of K

Formal Systems Laboratory
University of Illinois

September 2, 2017

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

1 Matching Logic

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

Add references.

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

Figure 1: The grammar of matching logic.

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 $\_*\_(\_+\_(x{:}Nat, y{:}Nat), z{:}Nat)$. More notational convenience and conventions will be introduced along the way as use them.

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 This is not true. Pattern φ contains quantifiers.
$\varphi_1 \mathbin{\texttt{+}} \varphi_2 = \varphi_1 +_{Nat} \varphi_2$	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.

2 A Calculus of Matching Logic

In this section, we define a matching logic theory $K = (S_K, \Sigma_K, A_K)$ as *the calculus of matching logic*, where S_K, Σ_K, and A_K are sets of sorts, symbols, and axioms, respectively.

2.1 Boolean Algebra

The matching logic theory of Boolean algebra is included in K, and the corresponding sort is named $\mathit{KBool} \in S_K$. Constructors of the sort *KBool* are two functional symbols

$$\mathit{Ktrue}\colon\ \to \mathit{KBool} \qquad \mathit{Kfalse}\colon\ \to \mathit{KBool}.$$

Common Boolean operators are defined as functional symbols with their corresponding axioms

$$\begin{aligned}
&\mathit{KnotBool}\colon \mathit{KBool} \to \mathit{KBool} && \mathit{KnotBool}(\mathit{Ktrue}) = \mathit{Kfalse}\\
&\mathit{KandBool}\colon \mathit{KBool} \times \mathit{KBool} \to \mathit{KBool} && \mathit{KnotBool}(\mathit{Kfalse}) = \mathit{Ktrue}\\
&\mathit{KorBool}\colon \mathit{KBool} \times \mathit{KBool} \to \mathit{KBool} && \mathit{KandBool}(\mathit{Ktrue}, b) = b\\
&\mathit{KimpliesBool}\colon \mathit{KBool} \times \mathit{KBool} \to \mathit{KBool} && \mathit{KandBool}(\mathit{Kfalse}, b) = \mathit{Kfalse}
\end{aligned}$$

The symbols *KorBool* and *KimpliesBool* are defined in terms of the symbols *KnotBool* and *KandBool* in the usual way

$$KorBool(b_1, b_2) = KnotBool(KandBool(KnotBool(b_1), KnotBool(b_2)))$$
$$KimpliesBool(b_1, b_2) = KorBool(KnotBool(b_1), b_2).$$

Notation 1. If b is a term pattern of sort *KBool*, then we will write just b instead of $b = Ktrue$ so that we can use Boolean expressions in any sort context.

Notation 2. To write Boolean expressions compactly, we adopt the following abbreviations if there is no confusion

$$\neg b \equiv KnotBool(b)$$
$$b_1 \wedge b_2 \equiv KandBool(b_1, b_2)$$
$$b_1 \vee b_2 \equiv KorBool(b_1, b_2)$$
$$b_1 \to b_2 \equiv KimpliesBool(b_1, b_2).$$

2.2 Strings

Introduce the matching logic theory for strings here.

2.3 Matching Logic Sorts and Symbols

The sort *KSort* is the sort for matching logic sorts. The only constructor of the sort *KSort* is the functional symbol:

$$Ksort \colon KString \to KSort.$$

The sort *KSymbol* is the sort for matching logic symbols. The only constructor of the sort *KSymbol* is the functional symbol:

$$Ksymbol \colon KString \to KSymbol.$$

2.4 Lists

Whenever we introduce a sort, say X, to S_K, we feel free to use *XList* as the sort of lists over X with the following symbols implicitly declared:

This is just a notation convention which allows us to have sorts like *KPatternList* and *KSortList* without defining each of them. We are NOT introducing any parametric modules here.

$$nilXList : \ \to XList$$
$$appendXList : XList \times XList \to XList$$
$$inXList : X \times XList \to KBool$$
$$XListAsX : X \to XList,$$

with axioms saying that $\mathit{appendXList}$ is associative and $\mathit{nilXList}$ is its identity. We adopt the following shorthands:

$$
\begin{aligned}
&\mathit{nil} \quad \text{as a shorthand of } \mathit{nilXList}\\
&\varphi_e \in \varphi_l \quad \text{as a shorthand of } \mathit{inXList}(\varphi_e, \varphi_l) = \mathit{Ktrue}\\
&\varphi_e \notin \varphi_l \quad \text{as a shorthand of } \mathit{inXList}(\varphi_e, \varphi_l) = \mathit{Kfalse}\\
&\mathit{appendXList}() \quad \text{as a shorthand of } \mathit{nil}\\
&\mathit{appendXList}(\varphi) \quad \text{as a shorthand of } \mathit{XListAsX}(\varphi)\\
&\mathit{appendXList}(\varphi_1, \ldots, \varphi_n) \quad \text{as a shorthand of}\\
&\qquad \mathit{appendXList}(\mathit{XListAsX}(\varphi_1),\\
&\qquad \mathit{appendXList}(\mathit{XListAsX}(\varphi_2),\\
&\qquad\qquad \ldots,\\
&\qquad \mathit{appendXList}(\mathit{XListAsX}(\varphi_n), \mathit{nil}))) \text{ when } n \geq 2.
\end{aligned}
$$

2.5 Matching Logic Patterns

The sort $\mathit{KPattern}$ is the sort for matching logic patterns. Constructors of the sort $\mathit{KPattern}$ are the following functional symbols:

$$
\begin{aligned}
&\mathit{Kvariable}\colon \mathit{KString} \times \mathit{KSort} \to \mathit{KPattern}\\
&\mathit{Kand}, \mathit{Kor}, \mathit{Kimplies}, \mathit{Kiff}\colon \mathit{KPattern} \times \mathit{KPattern} \times \mathit{KSort} \to \mathit{KPattern}\\
&\mathit{Knot}\colon \mathit{KPattern} \times \mathit{KSort} \to \mathit{KPattern}\\
&\mathit{Kapplication}\colon \mathit{KSymbol} \times \mathit{KPatternList} \to \mathit{KPattern}\\
&\mathit{Kexists}, \mathit{Kforall}\colon \mathit{KString} \times \mathit{KSort} \times \mathit{KPattern} \times \mathit{KSort} \to \mathit{KPattern}\\
&\mathit{Kequals}, \mathit{Kcontains}\colon \mathit{KPattern} \times \mathit{KPattern} \times \mathit{KSort} \times \mathit{KSort} \to \mathit{KPattern}\\
&\mathit{Ktop}, \mathit{Kbottom}\colon \mathit{KSort} \to \mathit{KPattern}.
\end{aligned}
$$

Notation 3. As a convention, we use b as KBool variables, x, y, z for $\mathit{KString}$ variables, s for KSort variables, σ for $\mathit{KSymbol}$ variables, and p, q, r for $\mathit{KPattern}$ variables.

2.6 Matching Logic Theories

WIP

There are also AST-related symbols included in Σ_K. For example, the symbol $\mathit{wellFormed}\colon \mathit{KPattern} \to \mathit{KBool}$ determines whether a pattern is well-formed (or more precisely, it determines whether an abstract syntax tree is a well-formed one of a pattern.), with axioms:

Provide axioms for all AST-related symbols.

$$\mathit{wellFormed}(\mathit{Kvariable}(x, s))$$

The symbol $\mathit{getSort} \in \Sigma_{\mathit{KPattern},\mathit{KSort}}$ takes a pattern and returns its sort. If the pattern is not well-formed, then $\mathit{getSort}$ returns $\bot_{\mathit{KSort}}$; otherwise, $\mathit{getSort}$

returns $Ksort(s)$ if the pattern has sort s. The symbol $getFvs\colon KPattern \to KPatternList$ collects all free variables in a pattern. The symbol $freshName\colon KPatternList \to KString$ generates a deterministic variable name that does not occur free in patterns in the argument. The symbol $Ksubstitute\colon KPattern \times KPattern \times KPattern \to KPattern$ takes a target pattern φ, a "find"-pattern ψ_1, and a "replace"-pattern ψ_2, and returns φ in which ψ_2 is substituted for ψ_1, denoted as $\varphi[\psi_2/\psi_1]$. All such AST-related symbols can be axiomatized in K. We take *Ksubstitute* as an example. The following axioms define *substitution*:

$$
\begin{aligned}
&Ksubstitute(r, q, r) = q\\
&Ksubstitute(Kand(p_1, p_2), q, r)\\
&\quad = Kand(Ksubstitute(p_1, q, r), Ksubstitute(p_2, q, r))\\
&Ksubstitute(Kor(p_1, p_2), q, r)\\
&\quad = Kor(Ksubstitute(p_1, q, r), Ksubstitute(p_2, q, r))\\
&\ldots\\
&Ksubstitute(Kexists(x{:}String, s, p), q, r)\\
&\quad = Kexists(freshName(p, q, r), s,\\
&\qquad\quad Ksubstitute((Ksubstitute(p, Kvariable(freshName(p, q, r), s),\\
&\qquad\qquad Kvariable(x{:}String, s), q, r))\\
&\ldots
\end{aligned}
$$

Side conditions can be defined as functional symbols from *KPattern* to *KBool*. For example, the symbol *syntactic* determines whether a pattern contains only variables and symbol applications. The symbol *ground* determines whether a pattern is variable-free, no matter free or bound. The symbol *groundSyntactic* determines whether a pattern is both syntactic and ground. They all can be easily defined in K. We will provide examples in later sections.

2.7 Theories

The calculus K contains sorts and symbols that are related to abstract syntax trees of matching logic theories. The sort $Signature \in S_K$ is the sort of matching logic signatures whose only constructor symbol is $signature\colon SortList \times SymbolList \to Signature$. The sort *Theory* is the sort of matching logic theories whose only constructor symbol is $theory\colon Signature \times KPatternList \to Theory$, which takes a signature and an axiom set as arguments.

2.8 Proof System

A proof system is a theorem generator. In K, the proof system of matching logic is captured by the functional symbol $deducible\colon KPattern \to KBool$, which returns *Ktrue* iff the argument pattern is a theorem. Given a matching logic pattern φ, we use $lift[\varphi]$ to denote its abstract syntax tree, where $lift[\_]$ is called the *lifting function* that maps object-patterns to their meta-representations in K.

It worths to point out that the lifting function $\mathit{lift}[\_]$ *cannot* be defined in K no matter what. It is purely a mathematical notation and is not part of the calculus. To see that, simply consider $\mathit{lift}[0]$ and $\mathit{lift}[x-x]$, where $0 = x - x$ but their ASTs are different:

$$\begin{aligned}
\mathit{lift}[0] &= \mathit{Kapplication}(\mathit{symbol}(\text{“}0\text{”}, \cdots), \ldots) \\
&\neq \mathit{lift}[x-x] \\
&= \mathit{Kapplication}(\mathit{symbol}(\text{“}_-_\text{”}, \cdots), \ldots)
\end{aligned}$$

This means that the following equational substitution deduction

$$\frac{\varphi_1 = \varphi_2}{\mathit{lift}[\varphi_1] = \mathit{lift}[\varphi_2]} \ (\text{WRONG})$$

does not hold. It is a strong evidence that $\mathit{lift}[\_]$ is not part of the logic.

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

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

Intuitively, $[\![\varphi]\!]$ means that "φ is deducible". Whenever there is an inference rule (axioms are considered as rules with zero premise)

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

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

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

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

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

as axioms to K. We sometimes denote the extended theory as $\mathit{lift}[T]$ and call it the *meta-theory for T*.

2.9 Faithfulness

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

Definition 4. The calculus K is said to be faithful for matching logic, if for any matching logic syntactic theory T and its meta-theory $\mathit{lift}[T]$,

φ is a theorem in T iff $[\![\varphi]\!]$ is a theorem in $\mathit{lift}[T]$, for any pattern φ.

Theorem 5. *The calculus K is faithful for matching logic.*

Proof. TBC. □

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

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

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

3 The Kore Language

We have shown K, a calculus for matching logic in which we can specify everything about matching logic and matching logic theories, such as whether a pattern is well-formed, what sort a patter has, which patterns are deducible, free variables, fresh variables generation, substitution, etc. The calculus K provides a universe of pattern ASTs and the sound and complete proof system of matching logic. On the other hand, it is usually easier to work at object-level rather than meta-level. Even if all reasoning in a matching logic theory T can be faithfully lifted to and conducted in its meta-theory *lift*$[T]$, it does not mean one should always do so.

The Kore language is proposed to define matching logic theories using the calculus K. At the same time, it also provides a nice surface syntax (syntactic sugar) to write object-level patterns. We will firstly show the formal grammar of Kore in Section 3.1, followed by some examples in Section 3.2. After that, we will introduce a transformation from Kore definitions to meta-theories as the formal semantics of Kore in Section **??**.

3.1 Syntax and Semantics of Kore

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

Variable       = VariableId:Sort
MetaVariable   = MetaVariableId::Sort

Pattern        = Variable | MetaVariable
```

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

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

Module          = module ModuleId
                    Sentences
                  endmodule
```

In Kore syntax, the backslash "\" is reserved for matching logic connectives and the sharp "`#`" is reserved for the meta-level, i.e., the K sorts and symbols. Therefore, the sorts *KBool*, *KString*, *KSymbol*, *KSort*, and *KPattern* in the calculus K are denoted as `#Bool`, `#String`, `#Symbol`, `#Sort`, and `#Pattern` in Kore respectively. Symbols in K are denoted in the similar way, too. For example, the constructor symbol $\mathit{Kvariable}\colon \mathit{KString} \times \mathit{KSort} \to \mathit{KPattern}$ is denoted as `#variable` in Kore.

A Kore module definition begins with the keyword `module` followed by the name of the module-being-defined, and ends with the keyword `endmodule`. The body of the definition consists of some *sentences*, whose meaning are introduced in the following.

The keyword `import` takes an argument as the name of the module-being-imported, and looks for that module in previous definitions. If the module is found, the body of that module is copied to the current module. Otherwise, nothing happens. The keyword `syntax` leads a *syntax declaration*, which can be either a *sort declaration* or a *symbol declaration*. Sorts declared by sort declarations are called *object-sorts*, in comparison to the five *meta-sorts*, `#Bool`, `#String`, `#Symbol`, `#Sort`, and `#Pattern`, in K. Symbols whose argument sorts and return sort are all object-sorts (meta-sorts) are called *object-symbols* (*meta-sorts*).

Patterns are written in prefix forms. A pattern is called an *object-pattern* (*meta-pattern*) if all sorts and symbols in it are object (meta) ones. Meta-symbols will be added to the calculus K, while object-sorts and object-symbols will not. They only serve for the purpose to parse an object pattern.

The keyword `axiom` takes a pattern and adds an axiom to the calculus K. If the pattern is a meta-pattern, it adds the pattern itself as an axiom. If the pattern φ is an object-pattern, it adds $[\![\varphi]\!]$ as an axiom to the calculus K.

Recall that we have defined the semantics bracket as

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

where φ is a pattern of the grammar in Figure 1. However, here in Kore we allow φ containing *meta-variables*. As a result, we modify the definition of the semantics bracket as

$$[\![\varphi]\!] \equiv \mathit{mvsc}[\varphi] \to (\mathit{deducible}\,(\mathit{lift}[\varphi]) = \mathit{true}),$$

where the lifting function $\mathit{lift}[\_]$ and the meta-variable sort constraint $\mathit{mvsc}[\_]$ are defined in Algorithm 1 and 2, respectively. Intuitively, meta-variables in an object-pattern φ are lifted to variables of the sort $\mathit{KPattern}$ with the corresponding sort constraints. For example, the meta-variable $x{::}s$ is lifted to a variable $x{:}\mathit{KPattern}$ in K with the constraint that $\mathit{getSort}(x{:}\mathit{KPattern}) = \mathit{sort}(s)$. The function $\mathit{mvsc}[\_]$ collects all such meta-variable sort constraint in an object-pattern is implemented in Algorithm 2.

Algorithm 1: Lifting Function $\mathit{lift}[\_]$

Input: An object-pattern φ.
Output: The meta-representation (ASTs) of φ in K

if φ *is* $x{:}s$ **then**
  Return $\mathit{variable}(x, \mathit{sort}(s))$
else if φ *is* $x{::}s$ **then**
  Return $x{:}\mathit{KPattern} \wedge (\mathit{sort}(s) = \mathit{getSort}(x{:}\mathit{KPattern}))$
else if φ *is* $\varphi_1 \wedge \varphi_2$ **then**
  Return $\mathit{Kand}(\mathit{lift}[\varphi_1], \mathit{lift}[\varphi_2]$
else if φ *is* $\neg\varphi_1$ **then**
  Return $\mathit{Knot}(\mathit{lift}[\varphi_1])$
else if φ *is* $\exists x{:}s.\varphi_1$ **then**
  Return $\mathit{Kexists}(x, \mathit{sort}(s), \mathit{lift}[\varphi_1])$
else if φ *is* $\sigma(\varphi_1, \ldots, \varphi_n)$ *and* $\sigma \in \Sigma_{s_1,\ldots,s_n,s}$ **then**
  Return $\mathit{Kapplication}(\mathit{symbol}(\sigma, (\mathit{Ksort}(s_1), \ldots, \mathit{Ksort}(s_n)), \mathit{Ksort}(s)), \mathit{lift}[\varphi_1], \ldots, \mathit{lift}[\varphi_n])$

Algorithm 2: Meta-Variable Sort Constraint Collection mvsc

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

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

3.2 Examples of Kore

Xiaohong: Add more examples and texts here.

The BOOL module.

```
module BOOL
  syntax Bool
  syntax Bool ::= true | false | notBool(Bool)
                | andBool(Bool, Bool) | orBool(Bool, Bool)
  axiom \or(true(), false())
  axiom \exists(X:Bool, \equals(X:Bool, true()))
  axiom \equals(andBool(B1::Bool, B2::Bool),
                andBool(B2::Bool, B1::Bool))
  axiom ... ...
endmodule
```

The BOOL module (desugared).

```
module BOOL
  axiom \equals(
    #true,
    #deducible(#or(#application(#symbol("true", #nilSort, #sort("Bool")),
                                #nilPattern),
                   #application(#symbol("false", #nilSort, #sort("Bool")),
                                #nilPattern))))
  axiom \equals(
    #true,
    #deducible(#exists("X", #sort("Bool"),
               #equals(#variable("X", #sort("Bool")),
                       #application(#symbol("true", #nilSort, #sort("Bool")),
                                    #nilPattern)))))
  axiom \implies(
    \and(\equals(#getSort(B1:Pattern), #sort("Bool")),
         \equals(#getSort(B2:Pattern), #sort("Bool"))),
    \equals(
      #true,
      #deducible(#equals(#application(#symbol("andBool",
                                              (#sort("Bool"), #sort("Bool"))
                                              #sort("Bool")),
                                      (B1:Patern, B2:Pattern)), ---- TODO
                         #application(#symbol("andBool",
                                              (#sort("Bool"), #sort("Bool"))
                                              #sort("Bool")),
                                      (B2:Patern, B1:Pattern))))))
  axiom ... ...
endmodule
```

The LAMBDA module

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

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