On the Effectiveness of Higher-Order Logic Programming in Language-Oriented Programming

Matteo Cimini(✉)

University of Massachusetts Lowell, Lowell, MA 01854, USA
matteo_cimini@uml.edu

Abstract. In previous work we have presented LANG-N-PLAY, a functional language-oriented programming language with languages as first-class-citizens. Language definitions can be bound to variables, passed to and returned by functions, and can be modified at run-time before being used. LANG-N-PLAY programs are compiled and executed in the higher-order logic programming language λProlog. In this paper, we describe our compilation methods, which highlight how the distinctive features of higher-order logic programming are a great fit in implementing a language-oriented programming language.

Keywords: Higher-order logic programming · Language-oriented programming · Functional programming

1 Introduction

Language-oriented programming [8,14,31] is a paradigm that has received a lot of attention in recent years. Behind this paradigm is the idea that different parts of a programming solution should be expressed with different problem-specific languages. For example, programmers can write JavaScript code for their web application and enjoy using JQuery to access DOM objects in some parts of their code, and WebPPL to do probabilistic programming in other parts [16]. To realize this vision, *language workbenches* have emerged as sophisticated tools to assist programmers with the creation, reuse and composition of languages.

Languages as first-class citizens [7] is an approach to language-oriented programming that advocates that language definitions should have the same status as any other expression in the context of a general-purpose programming language. In this approach language definitions are *run-time values*, just like integers, for example, and they can be the result of computations, bound to variables, passed to and returned by functions, and inserted into lists, to name a few possibilities.

LANG-N-PLAY [6,7] is a functional language-oriented programming language with languages as first-class citizens. LANG-N-PLAY is implemented in a combination of OCaml and the higher-order logic programming language λProlog [24].

The core of the language implementation is, however, an interpreter written in λProlog. Specifically, LANG-N-PLAY programs are compiled into λProlog terms and executed with such interpreter.

To implement language-oriented programming operations, the features of higher-order logic programming have proved to be exceptionally fitting. In particular, formulae as first-class citizens make it easy to have a run-time data type for languages and implement operations that manipulate languages at run-time, including switching evaluation strategies on demand (call-by-value vs call-by-name). Furthermore, hypothetical reasoning [15] (i.e. deriving implicative goals)[1] makes it easy to execute programs with arbitrary languages defined by programmers, as well as switch from a language to another during computation.

Goal of the Paper. Our goal is to demonstrate that higher-order logic programming can be a great fit for implementing language-oriented systems. To this aim, we describe our compilation methods and highlight how the distinctive features of higher-order logic programming have been a natural fit in this context.

Ultimately, LANG-N-PLAY allows for non-trivial language-oriented programming scenarios, and yet its interpreter is 73 lines of λProlog code. This is remarkable in the context of language-oriented systems.

Roadmap of the Paper. Section 2 reviews higher-order logic programming as adopted in λProlog. Section 3 gives a general overview of the implementation of LANG-N-PLAY before diving into specific aspects. Section 4 discusses our implementation of (programmer-defined) language definitions. Section 5 covers our implementation of the LANG-N-PLAY operations that manipulate languages. Section 6 provides details on our implementation w.r.t. using languages to execute programs. Section 7 covers the scenario of switching strategies at run-time. Section 8 discusses related work, and Sect. 9 concludes the paper.

2 Higher-Order Logic Programming

λProlog is a flagship representative of higher-order logic programming languages [24]. This section reviews its features. We do not give a complete account of λProlog. Instead, we discuss the features that play a role in the next sections. λProlog extends Prolog with the following elements: types, formulae as first-class citizens, higher-order abstract syntax, and hypothetical reasoning.

Typed Logic Programming. λProlog programs are equipped with a signature that defines the entities that are involved in the program. Programs must follow the typing discipline that is declared in this signature or they would be rejected. For example, if we were to implement a simple language with numbers and additions, we would have the following declarations.

[1] To remain in line with λProlog terminology we use the terms *hypothetical reasoning* throughout this paper, see [24].

```
kind typ type.
kind expression type.

type int typ.
type zero -> expression.
type succ expression -> expression.
type plus expression -> expression -> expression.
```

The keyword `kind` declares the entities in the program. The keyword `type` is used to specify how to create terms of such entities.

As in Prolog, the computation takes place through the means of logic programming rules. Logic programming rules can derive formulae, which are built with predicates. Predicates, as well, must be declared with a type in λProlog. For example, a type checking relation and a reduction relation can be declared as follows.

```
type typeOf expression -> typ -> prop.
type step expression -> expression -> prop.

typeOf zero int.
typeOf (succ E) int :-  typeOf E int.

  ... reduction rules, here omitted ...
```

The keyword `prop` denotes that `typeOf` and `step` build a formula when applied to the correct type of arguments.

Formulae as First-Class Citizens. λProlog extends Prolog also in that formulae can be used in any well-typed context. For example, below we intentionally split our type checker in an unusual way.

```
type getFormula expression -> prop -> prop.
type check expression -> prop.

getFormula zero true.
getFormula (succ E) (typeOf E int).

check E :- getFormula E F, F.
```

The predicate `getFormula` takes two arguments. The first is an expression and is an input, and the second is a proposition and is an output. The predicate `getFormula` returns the formula we should check to establish that the term is well-typed (the output type is ignored for the sake of this example). This example shows that formulae can be arguments. Furthermore, after `check` calls `getFormula` to retrieve the formula `F`, this formula can be used as a premise in the rule, as shown in the last line.

Higher-Order Abstract Syntax (HOAS). HOAS is an approach to syntax in which the underlying logic can appeal to a native λ-calculus for modeling aspects related to binding [25]. Suppose that we were to add the operators of the λ-calculus, we would define the following.

```
type abs (expression -> expression) -> expression.
type app expression -> expression -> expression.

step (app (abs R) V) (R V) :- value V.
```

The argument of `abs` is an abstraction from an expression to an expression. To model the identity function we write (`abs x\ x`), where the highlighted part of this term points out the syntax used by λProlog for writing HOAS abstractions. In the beta-reduction rule above we have that `R` is an abstraction and therefore we can use it with a HOAS application (`R V`) to produce a term. λProlog takes care of performing the substitution for us.

Hypothetical Reasoning. λProlog also extends Prolog with hypothetical reasoning [24] (i.e. deriving implicative goals [15]). To appreciate this feature consider the following logic program.

```
flyTo london nyc.
flyTo chicago portland.
connected X X.
connected X Z :- flyTo X Y, connected Y Z.
```

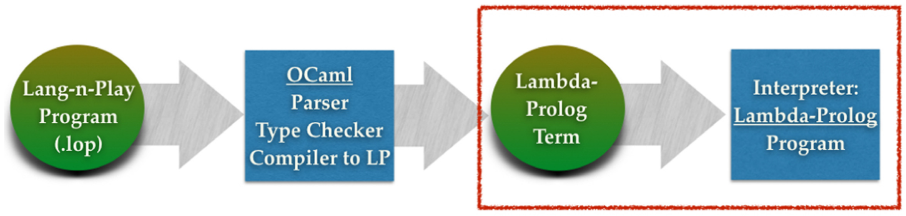
The city `london` and `portland` are not connected. However, in λProlog we can write the formula:

```
connected nyc chicago => connected london portland
```

This formula asks: "Were `nyc` connected to `chicago`, would `london` be connected to `portland`?". At run-time the query `connected london portland` is interrogated in the logic program in which the fact `connected nyc chicago` is added.

3 Basic Overview of Lang-n-Play

LANG-N-PLAY is a functional language-oriented programming language [7]. Programmers can define their own languages and use them to execute programs. LANG-N-PLAY is implemented partly in OCaml and partly in λProlog. Precisely, the following is the architecture of LANG-N-PLAY.



Programs are parsed and type checked in OCaml. These two aspects are not discussed in the paper because they do not play a role in our message about the effectiveness of higher-order logic programming.

Next, LANG-N-PLAY programs are compiled into λProlog. The interpreter of LANG-N-PLAY programs is a λProlog logic program. Our OCaml compilation produces the λProlog term that represents the LANG-N-PLAY program, and gives it to this interpreter. Programmers do not launch the interpreter manually. Our OCaml code is interfaced to the ELPI interpreter of λProlog through the ELPI OCaml package [9], and loads an external file that contains the interpreter written in λProlog.

In this paper we discuss the part that is enclosed in the red rectangle, as it is the core of the implementation of LANG-N-PLAY, and highlights how higher-order logic programming can accommodate the features of LANG-N-PLAY.

Our interpreter defines a kind for LANG-N-PLAY expressions called `expLO`, and defines two relations for reduction steps and detecting values.

```
type stepLO expLO -> expLO -> prop.
type valueLO expLO -> prop.
```

The suffix `LO` in the names of operators and relations are meant to recall that we are in a language-oriented language. We introduce the elements of `expLO` as we encounter them in the next sections. There are a number of aspects of LANG-N-PLAY that we do not discuss in this paper. For example, although LANG-N-PLAY includes common features of functional programming such as booleans, if-then-else, lists, `letrec`, and `import`, we omit discussing them because they are not relevant to this paper.

4 Defining Languages

LANG-N-PLAY provides syntax for defining languages. Below is the example of a definition of a language with lists and an operator `elementAt` which accesses an element of a list based on its position (position 0 is the first element).

```
{!
Type T ::= int | (list T),
Expression e ::= zero | (succ e) | nil | (cons e e)
                 | (elementAt e e),
Value v ::= zero | (succ v) | nil | (cons v v),
Context C ::= (succ E) | (cons C e) | (cons v C)
              | (elementAt C e) | (elementAt v C),
Environment Gamma ::= [x : T],
Relation ::= Gamma |- e : T | e --> e,
StartingCall ::= empty |- e : T | e --> e.

Gamma |- x : T <== x : T in Gamma,
Gamma |- zero : int,
Gamma |- (succ e) : int <== Gamma |- e : int,
Gamma |- nil : (list T),
Gamma |- (cons e1 e2) : (list T) <==
         Gamma |- e1 : T /\ Gamma |- e2 : (list T),
Gamma |- (elementAt e1 e2) : T <==
         Gamma |- e1 : int /\ Gamma |- e2 : (list T),
(elementAt zero (cons V1 V2)) --> V1,
(elementAt (succ V) (cons V1 V2)) --> (elementAt V V2)
!}
```

Languages are defined within `{! ... !}`, as in lines 1–22. LANG-N-PLAY makes use of a domain-specific language that closely resembles the way researchers define and share languages in operational semantics. (The Ott language achieved the same effect over ten years ago [28], and we adopt a similar syntax).

As typical, languages define a grammar (lines 2–10) and an inference system (lines 12–21). An inference system may define a type system and a reduction semantics.

The syntax for defining grammars is quite standard. As in many language workbenches [8,14,31], language definitions can use lists with `[ ... ]`. For example, `Environment Gamma ::= [x : T]` means `Gamma` is a list of formulae of that shape. LANG-N-PLAY provides the operation `in` for testing membership on lists, as in line 12. Furthermore, there are two special grammar categories, `Relation` and `Starting Call`. The former simply declares relations, and the latter informs LANG-N-PLAY on how to call the type checker and the evaluator.

The syntax for defining inference systems is also rather familiar to operational semantics practitioners. Perhaps, the biggest departure is that the horizontal line of an inference rule is replaced with an inverse implication `<==` that can be read *"provided that"*, and we use an explicit syntactic and-operator when we have multiple premises.

The language definition above serves as our running example throughout the paper. We refer to the lines 2–21 as *listLines* in what follows.

4.1 λProlog Implementation of Language Definitions

It is rather natural to compile language definitions such as the one above into λProlog because operational semantics is based on inference systems. These, in turn, map naturally to logic programming rules. For example, the language above compiles to the following (we show only an excerpt).

```
typeOf nil (list T).
typeOf (cons E1 E2) (list T) :- typeOf E1 T,
                                typeOf E2 (list T).
typeOf (elementAt E1 E2) T :- typeOf E1 int,
                              typeOf E2 (list T).
step (elementAt zero (cons V1 V2)) V1 :- value V1,
                                         value V2.
step (elementAt (succ V) (cons V1 V2)) (elementAt V V2)
                                   :- value V1, value V2.
```

The fact that language definitions map well to higher-order logic programs has been previously demonstrated with the work of Twelf [26], and λProlog [24].

Since LANG-N-PLAY handles programmer-defined languages that may be manipulated at run-time, we accommodate languages with an internal representation. In particular, the language above is represented with a list of formulae with the following operator:

```
type language (list prop) -> expLO.
```

Therefore, the language *listLines* is compiled as follows.

```
 1  language
 2  [
 3  typeOf nil (list T) ;
 4  typeOf (cons E1 E2) (list T) :- typeOf E1 T,
 5                                  typeOf E2 (list T) ;
 6  typeOf (elementAt E1 E2) T :- typeOf E1 int,
 7                                typeOf E2 (list T) ;
 8  step (elementAt zero (cons V1 V2)) V1 :- value V1,
 9                                           value V2 ;
10  step (elementAt (succ V) (cons V1 V2)) (elementAt V V2)
11                                     :- value V1, value V2 ;
12  value nil ;
13  value (cons V1 V2) :- value V1, value V2 ;
14  step (elementAt E1 E2) (elementAt E1' E2) :- step E1 E1';
15  step (elementAt V1 E2) (elementAt V1 E2') :- step E2 E2';
16  ... the rest of contextual reduction rules ...
17  ]
```

This is our run-time representation for languages. Notice that since we need only the reduction rules to execute programs we compile inference rules only (not grammar), with the exception of values and evaluation contexts, which are turned into rules.

We shall refer to the list of elements at lines 3–16 as *listsInLP*.

5 Operations on Languages

LANG-N-PLAY provides a handful of operations on language definitions. Below we discuss the following operations: let-binding, union of languages, functions on languages, and removal of rules.

Let-Binding. LANG-N-PLAY can bind a language definition to a variable in typical ML-style, as in

```
let lists = {!
  ... listLines ...
!} in lists
```

Therefore, our interpreter includes a let-operation and its reduction semantics.

```
type letLO expLO -> (expLO -> expLO ) -> expLO.
stepLO (letLO V R) (R V) :- valueLO V.
```

The code above is then compiled to

```
letLO (language [ listsInLP ]) (lists\ lists)
```

which reduces to `(language [ listsInLP ])` in one step. (This example also shows that languages can be the result of computations.)

Language Union. Another operation of LANG-N-PLAY is language union, performed with the binary operator `U`. For example, notice that the language for lists is unsafe: if we called `elementAt` asking for the 4-th element of a list that contains only 2 elements we would get stuck, because there is no specified behavior for that case. We can add reduction rules that cover those cases as follows. (We refer to the language created by the union operation below as `safeLists`).

```
lists U {!
   Expression e ::= myError,
   Error er ::= myError,
   (elementAt zero nil) --> myError,
   (elementAt (succ V) nil) --> myError,
 !}
```

This code adds the error to the language and adds appropriate reduction rules. Our interpreter includes the language union operation and its reduction semantics.

```
type unionLO expLO -> expLO -> expLO.
stepLO
     (unionLO (language Rules1) (language Rules2))
     (language Result)
     :- append Rules1 Rules2 Result.
```

where `append` is a polymorphic list append defined in the interpreter (straightforward and here omitted). The union operation above is compiled as

```
1  unionLO
2      (language [ listsInLP ])
3      (language [ step (elementAt zero nil) myError ;
4                  step (elementAt (succ V) nil) myError])
```

which reduces to `(language [ listsInLP + rules in lines 3-4 ])` in one step.

Removal of Rules. LANG-N-PLAY includes an operation for removing rules from a language. For example, the union above adds two extra rules but the sole rule `step (elementAt V nil) myError :- value V` would be sufficient.

To modify `safeLists` and create the more compact language that has only one rule we can execute the following LANG-N-PLAY program.

```
1  (remove
2       (elementAt zero nil) --> myError)
3       from (remove (elementAt (succ V) nil) --> myError
4            from safeLists)
5  ) U {! (elementAt V nil) --> myError !}
```

The operation `remove` takes in input a rule and a language, and returns a language. This code removes one of the rules from `safeLists` at lines 3 and 4. The language so produced is then used in the removal of the other rule at lines 1–3. Line 5 adds the safe reduction rule for `elementAt` with a union operation.

Our interpreter includes the rule removal operation and its reduction semantics.

```
type removeLO prop -> expLO -> expLO.
stepLO (removeLO Formula (language Rules))
       (language Result)
       :- listRemove Formula Rules Result.
```

where `listRemove` is a polymorphic predicate that matches elements of a list with a given element and removes the element if the match succeeds. This predicate is also defined in the interpreter (straightforward and here omitted).

The remove operations above, excluding the union at line 5, are compiled as

```
 1  removeLO
 2      (step (elementAt zero nil) myError)
 3      (removeLO
 4         step (elementAt (succ V) nil) myError :- value V
 5         (language [ listsInLP +
 6           step (elementAt zero nil) myError ;
 7           step (elementAt (succ V) nil) myError :- value V
 8                   ]
 9         )
10      )
```

which reduces to `(language [ listsInLP ])` in two steps.

λProlog grants us a powerful equality on formulae and the removal operation is far from performing a textual match on the rule to remove. For example, the following two rules are equal in λProlog.

```
(*)    step (elementAt (succ MyVar) nil) myError :- value MyVar
                                   =
        step (elementAt (succ V) nil) myError :- value V
```

Therefore, we obtain the same results if we used the formula `(*)` at line 4. Thanks to λProlog, formulae are up-to renaming of variables and alpha-equivalence of HOAS abstractions.

Functions on Languages. As typical in programming languages, we often would like to pack instructions in functions for the sake of abstraction. LANG-N-PLAY provides functions on languages. For example, instead of performing language union inline we can create a function that adds the desired safety checks, as in

```
let addSafeAccess mylan =
mylan U {!  Expression e ::= myError,
            Error er ::= myError,
            (elementAt V nil) --> myError,
        !}
 in (addSafeAccess lists)
```

Our interpreter includes abstractions and applications, as well as their reduction semantics.

```
type absLO (expLO -> expLO ) -> expLO.
type appLO expLO -> expLO -> expLO.
stepLO (appLO (absLO R) V) (R V) :- valueLO V.
```

LANG-N-PLAY compiles the let-binding above in the following way.

```
letLO
  (addSafeAccess\
    (appLO addSafeAccess (language [... listsInLP ...])))
  (absLO mylan\
    (unionLO
      mylan
      (language [
         step (elementAt (succ V) nil) myError :- value V
                ])
    )
  )
```

This program reduces the `letLO` operation in one step and we obtain

```
(appLO
   (absLO mylan\ unionLO mylan
      (language [
         step (elementAt (succ V) nil) myError :- value V
                ])
   (language [... listsInLP ...])))
```

In turn, this program reduces in one step to

```
unionLO
  (language [... listsInLP ...])))
  (language [
         step (elementAt (succ V) nil) myError :- value V
            ])
```

which produces the expected language in one step.

6 Executing Programs and Language Switch

Of course, languages can be used to execute programs. The code below shows the LANG-N-PLAY expression to do so. (For readability, we use standard notation for numbers and lists rather than sequences of `succ` and `cons`).

```
{! ... listsLines ... !}> elementAt 1 [1,2,3]
```

We call this type of expression *program execution* and is of the form *language* > *program*. The program above returns a value together with the language with which it has been computed:

`Value = 2 in {! listsLines !}.`

Our interpreter includes the operation for executing programs and its reduction semantics.

```
type execLO expLO -> program -> expLO.

stepLO (execLO (language Language) Prg)
       (execLO (language Language) Prg')
       :- (Language => (step Prg Prg')).
```

In the declaration at the top, `program` is the kind for programs. Intuitively, elements of `program` are S-expressions, i.e. a top-level operator followed by a series of arguments, which, too, can be programs. Notice that the language argument of `execLO` (first argument) is an expression. Although above we have explicitly written the language to be used, that component can be an expression that evaluates to a language, for example as in

```
lists> elementAt 1 [1,2,3], or
(addSafeAccess lists)> elementAt 1 [1,2,3]
```

The reduction rule for `execLO` deserves some words. The key idea is that we use hypothetical reasoning. In Sect. 2 we have seen that we can use this feature to temporarily add facts and run an augmented logic program. Above, instead, we do not add a fact but a list of formulae `Language`. Moreover, this list of formulae is not a list of facts but a list of rules (rules such as `step (elementAt zero (cons V1 V2)) V1 :- value V1, value V2.`). This has the effect of augmenting the logic program that we are currently running (which is our LANG-N-PLAY interpreter!) with new rules. In particular, these rules define the operational semantics of the language that we need to execute. The interpreter then interrogates (`step Prg Prg'`) to compute the step from these new rules[2].

For example, the code above compiles to

```
execLO (language [... listsInLP ...]) (elementAt 1 [1,2,3])
```

The current logic program (that is, our LANG-N-PLAY interpreter) is augmented with the rules *listsInLP* and we execute the query
(`step (elementAt 1 [1,2,3]) Prg'`).

This produces `Prg'` = (`element 0 [2,3]`). (Notice that the query asks for one step). LANG-N-PLAY keeps executing until a value is produced. Therefore at the second step we run the query (`step (elementAt 0 [2,3]) Prg'`). This query returns the result `Prg' = 2`, which our interpreter detects as a value. The execution of programmer-defined languages and the execution of LANG-N-PLAY operations are never confused because the former makes use of the predicate `step` and the latter makes use of the predicate `stepLO`.

The way our interpreter recognizes that we have obtained a value is through the predicate `valueLO`, which our interpreter defines with

```
valueLO (execLO (language Language) Prg)
                              :- (Language => (value Prg)).
```

[2] We are guaranteed that the rules use the predicate `step` for reductions because the OCaml part of LANG-N-PLAY (see the figure on Sect. 3, page 4) specifically generates the λProlog term to use `step`. Similarly for `value`.

Let us recall that once *listsInLP* is loaded in the current logic program it also contains the rules that define the values of the loaded language, defined with the predicate `value`. These rules are, for example, `value zero`. and `value succ V :– value V`., and so on. Then we run the query (`value Prg`) to detect if `Prg` is a term that the loaded language defines as a value.

Language Switch. LANG-N-PLAY also allows for switching languages at run-time. Consider the following program.

```
let pairs =  {!
 Type T ::= (times T T),
 Expression e ::= pair e e | fst e | snd e,
 Value v ::=  (pair v v),
 Context C ::=  (pair C e) | (pair v C)
              | (fst C) | (snd C),
 Gamma |- (pair e1 e2) : (times T1 T2) <==
                   Gamma |- e1 : T1 /\ Gamma |- e2 : T2,
 Gamma |- (fst e) : T1 <==
                    Gamma |- e : (times T1 T2),
 Gamma |- (snd e) : T2 <==
                    Gamma |- e : (times T1 T2),
 (fst (pair V1 V2)) --> V1,
 (snd (pair V1 V2)) --> V2
 !} in
 lists> elementAt (pairs> fst (pair 1 0)) [1,2,3]
```

This code defines a language with pairs. Afterwards, it makes use of the list language to perform a list access. However, the first argument of `elementAt` (the position) is computed by executing another program in another language. In particular, the position is computed by executing `fst (pair 1 0)` in the language `pairs`.

This program returns the following value (recall that position `1` asks for the second element of the list).

```
          Value = 2 in {! listsLines !}.
```

Implementing this language switch is easy with higher-order logic programming. When we execute

```
execLO (language [... listsInLP ...])
       (elementAt
           (execLO (language [ rules of pairs ])
                   (fst (pair 1 0)))
           [1,2,3])
```

The interpreter adds the rules *listsInLP* and evaluates the program that starts with `elementAt`. When the interpreter encounters another `execLO` it applies the same reduction rule of `execLO` that we have seen. This has the effect of adding the language with pairs on top of the language with lists. This augmented language is then used to evaluate `fst (pair 1 0)` with the query `(step (fst (pair 1 0)) Prg')`. The nested `execLO` detects when `fst (pair 1 0)` evaluates to a value in the way described above, that is, the query `(value 1)` succeeds. At this point, `execLO` simply leaves the value it has just computed in the context in which it has been executed. Therefore, `elementAt` simply continues with the value `1` as first argument, oblivious as to how this value has been computed.

Remark on the Semantics of Language Switch. The semantics of language switch is such that the current language is *extended* with new rules. Therefore, the switch does not replace a language with a completely different language. The semantics we adopt is in 1-1 correspondence with the semantics of hypothetical reasoning. We believe that this facilitates writing correct programs because the child language must at least share some values with the parent language, as the nested computation leaves a value in the context of the parent. Therefore, this value must be understood by the parent language.

Notice that the overall result is 2 in the language {! *listsLines* !} that does not contain pairs. Indeed, pairs has been added and then removed by λProlog after the nested execLO has finished.

7 Valuehood Abstractions

Strategies play a central role in computing. Examples of notable strategies are call-by-value and call-by-name in the λ-calculus. In LANG-N-PLAY we can define the λ-calculus in a way that allows the strategy to be chosen at run-time. We do so with valuehood abstractions:

```
let lambda vh : strategy =
    {! Expression e ::= (abs @x e) | (app e e),
       Value v ::= (abs @x e),
       Context C ::= (app C e),
       Environment Gamma ::= [x : T],
       (app (abs @x e)  vh ) --> e[ vh /x],
    !}
```

Here, lambda is not a language definition. It is, instead, a valuehood abstraction. This is a function that takes in input a kind of expression called strategy. The variable vh is bound in the body of lambda and can appear as a variable in inference rules, as highlighted. The meaning is that it will be discovered later whether the inference rule fires when the variable is a value or an ordinary expression. The two strategies are represented with the constants EE and VV, respectively. The application (lambda EE) returns the language with the reduction rule (app (abs @x e) e2) --> e[e2/x], which fires irrespective of whether e2 is a value or not, in call-by-name style. The application (lambda VV), instead, return the language with rule (app (abs @x e) v) --> e[v/x], which fires when the argument is a value, in call-by-value style.

To realize this, we take advantage of formulae as first-class citizens. The compilation of lambda is:

```
absLO vh\
language [step (app (abs R) ARG) (R ARG) :- (vh ARG)]
```

The variable vh is a HOAS abstraction from terms to formulae. Intuitively, when we pass EE we want to say that the premise (vh ARG) should not add any additional conditions. We do so by compiling EE to the function (x\ true). The application (lambda EE) is then compiled into an ordinary application appLO.

For (`lambda EE`), after the parameter passing we end up with

```
 language [step (app (abs R) ARG) (R ARG) :- ((x\ true) ARG)]
```

which λProlog converts to

```
language [step (app (abs R) ARG) (R ARG) :- true]
```

When we pass `VV`, instead, we want to say that the premise `(vh ARG)` should be satisfied only so long that `ARG` is a value. To do so, we compile the constant `VV` to `(x\ value x)`. When we execute (`lambda VV`) we end up with

```
language [step (app (abs R) ARG) (R ARG) :- ((x\ value x) ARG)]
```

which λProlog converts to

```
 language [step (app (abs R) ARG) (R ARG) :- value ARG].
```

Therefore, we place a new premise to check that `ARG` is a value.

8 Related Work

The main related work is the vision paper [7]. We have used a variant of the example in [7] to demonstrate our compilation methods. We have also addressed `remove` with a different example, and the example on language switch is new.

There are two main differences between [7] and this paper.

- [7] proposes the approach of languages as first-class citizens and exemplifies it with an example in LANG-N-PLAY. [7] does not discuss any implementation details of LANG-N-PLAY. On the contrary, this paper's focus is entirely on the implementation mechanisms that we have adopted and, most importantly, this paper demonstrates that the distinctive features of higher-order logic programming are a great fit for language-oriented systems.
- This paper extends the work in [7] by adding language switches.

The K framework is a rewrite-based executable framework for the specification of programming languages [27]. Differently from LANG-N-PLAY, the K framework does not offer language-oriented programming features such as language switch and the updating of languages at run-time. On the other hand, K has been used to define real-world programming languages such as C [10] and Java [2], to name a few, while LANG-N-PLAY has not. The style of language definitions in LANG-N-PLAY is that of plain operational semantics. Therefore, defining real-world programming languages can become a cumbersome task, and we have not ventured into that task yet. An interesting feature of the K framework is that it automatically derives analyzers and verifiers from language specifications [29]. This work inspires us towards adding similar features to LANG-N-PLAY.

A number of systems have been created to support language-oriented programming [3,17,18,21,30]. Features such as language switch and the updating of languages at run-time are rather sophisticated in this research area, and not many systems offer these features [4,12,19,20]. However, some language workbenches do provide these functionalities, such as Racket [13], Neverlang [30] and Spoofax [18]. Racket provides special syntax for defining languages and its implementation is based on macros. Language definitions are macro-expanded into Racket's core functional language. Spoofax and Neverlang have an internal ad hoc representation of languages. Languages can be accessed and updated at run-time in Neverlang, for example, making use of the run-time loading features of the JVM to add parts to languages represented as objects. To our knowledge, LANG-N-PLAY is the only language-oriented system that is implemented in *higher-order* logic programming.

LANG-N-PLAY is *not* more expressive than other systems. The goal of this paper is not to provide a system that surpasses the state of the art. LANG-N-PLAY is also a young tool (2018) compared to the mentioned systems, some of which boast decades of active development. The goal of our paper is, instead, about the effectiveness of higher-order logic programming in language-oriented programming. It is hard to compare our implementation to others at a quantitative level (lines of code) because systems such as Racket, Neverlang and Spoofax are very large and mature systems that offer all sorts of language services. We were not able to single out an isolated meaningful part of these systems to compare with our interpreter. Nonetheless, we believe that it is generally remarkable that we could implement the interpreter of a full language-oriented programming language with sophisticated features in 73 lines of code.

9 Conclusion

This paper describes the compilation methods that we have adopted in the implementation of LANG-N-PLAY, and provides evidence that high-order logic programming can be a great fit for implementing language-oriented systems.

The following aspects of higher-order logic programming have been particularly fitting:
(We add numbers, as some features have been helpful in more than one way).

- *Formulae as first-class citizens #1:* List of formulae naturally models language definitions in operational semantics, providing a readily available run-time data type. This makes it easy to implement operations that manipulate languages (such as union and rule removal) during execution.
- *Formulae as first-class citizens #2:* It models naturally the switch of evaluation strategy at run-time. This is thanks to the fact that we can pass premises to rules. These premises may or may not add new conditions under which existing rules can fire.
- *Hypothetical reasoning #1:* It naturally models the execution of a program with a given operational semantics, possibly created at run-time.
- *Hypothetical reasoning #2:* It naturally models the switch from executing a program using a language to executing another program using an extension of that language.

In the future, we would like to strengthen our message by implementing further operations on languages using high-order logic programming. We would like to implement operations such as language unification and restriction [11], grammar inheritance, language embedding, and aggregation from the Manticore system [21], and renaming and remapping from Neverlang [30]. There is notable work in [5,22,23] on inferring the dependencies of languages, which we also plan to implement.

Some systems compile languages and programs into proof assistants. For example, Ott compiles into Coq, HOL and Isabelle [28], so that users can carry out proofs in these systems. Currently, LANG-N-PLAY compiles into λProlog solely to execute programs. However, a subset of λProlog is also the specification language of the proof assistant Abella [1]. In the future, we would like to explore the verification of LANG-N-PLAY programs after compilation to λProlog/Abella code.

We point out that language workbenches offer a variety of editor services among syntax colouring, highlighting, outlining, and reference resolution, to name a few. Currently, LANG-N-PLAY is not equipped with a comfortable IDE. Inspired by the work on language workbenches, we would like to improve the usability of our system.

References

1. Baelde, D., et al.: Abella: a system for reasoning about relational specifications. Journal of Formalized Reasoning **7**(2) (2014). https://doi.org/10.6092/issn.1972-5787/4650. http://jfr.unibo.it/article/download/4650/4137
2. Bogdanas, D., Rosu, G.: K-Java: a complete semantics of Java. In: Proceedings of the 42nd Symposium on Principles of Programming Languages, pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
3. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the gemoc studio (tool demo). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering SLE 2016, pp. 84–89. ACM, New York (2016)
4. van den Brand, M.G.J., et al.: The ASF+SDF meta-environment: a component-based language dvelopment environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45306-7_26
5. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Modeling language variability with reusable language components. In: Proceedings of the 22nd International Systems and Software Product Line Conference SPLC 2018. ACM, New York (2018)
6. Cimini, M.: Lang-n-play: a functional programming language with languages as first-class citizens (2018). https://github.com/mcimini/lang-n-play
7. Cimini, M.: Languages as first-class citizens (vision paper). In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering SLE 2018, pp. 65–69. ACM, New York (2018). https://doi.org/10.1145/3276604.3276983
8. Dmitriev, S.: Language oriented programming: the next programming paradigm (2004). http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf
9. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: fast, embeddable, \lambda prolog interpreter. In: Proceedings of the Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, 24–28 November 2015, pp. 460–468 (2015). https://doi.org/10.1007/978-3-662-48899-7_32

10. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th Symposium on Principles of Programming Languages, pp. 533–544. ACM (2012). https://doi.org/10.1145/2103656.2103719
11. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: LDTA 2012, pp. 7:1–7:8. ACM, New York (2012)
12. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: library-based syntactic language extensibility. SIGPLAN Not. **46**(10), 391–406 (2011). https://doi.org/10.1145/2076021.2048099. http://doi.acm.org/10.1145/2076021.2048099
13. Flatt, M., PLT: reference: racket. Technical report PLT-TR-2010-1. PLT Design Inc. (2010). https://racket-lang.org/tr1/
14. Fowler, M.: Language workbenches: the killer-app for domain specific languages? (2005). http://www.martinfowler.com/articles/languageWorkbench.html
15. Gabbay, D., Reyle, U.: N-prolog: an extension of prolog with hypothetical implications I. J. Logic Program. **1**(4), 319–355 (1984). http://www.sciencedirect.com/science/article/pii/0743106684900293
16. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages (2014). http://dippl.org. Accessed 10 Feb 2020
17. JetBrains: JetBrains MPS - Meta Programming System. http://www.jetbrains.com/mps/
18. Kats, L.C.L., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. In: OOPSLA, vol. 45, pp. 444–463. ACM, New York, October 2010. http://doi.acm.org/10.1145/1932682.1869497
19. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In: Constantopoulos, P., Mylopoulos, J., Vassiliou, Y. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61292-0_1
20. Kienzle, J., et al.: Concern-oriented language development (COLD): fostering reuse in language engineering. Comput. Lang. Syst. Struct. **54**, 139–155 (2018)
21. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. **12**(5), 353–372 (2010). https://doi.org/10.1007/s10009-010-0142-1
22. Kühn, T., Cazzola, W., Olivares, D.M.: Choosy and picky: configuration of language product lines. In: Proceedings of the 19th International Conference on Software Product Line SPLC 2015, pp. 71–80. ACM, New York (2015). https://doi.org/10.1145/2791060.2791092. http://doi.acm.org/10.1145/2791060.2791092
23. Méndez-Acuña, D., Galindo, J.A., Degueule, T., Combemale, B., Baudry, B.: Leveraging software product lines engineering in the development of external DSLs: a systematic literature review. Comput. Lang. Syst. Struct. **46**, 206–235 (2016). https://doi.org/10.1016/j.cl.2016.09.004
24. Miller, D., Nadathur, G.: Programming with Higher-Order Logic, 1st edn. Cambridge University Press, New York (2012)
25. Pfenning, F., Elliott, C.: Higher-order abstract syntax. SIGPLAN Not. **23**(7), 199–208 (1988). https://doi.org/10.1145/960116.54010
26. Pfenning, F., Schürmann, C.: System description: twelf—a meta-logical framework for deductive systems. CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_14
27. Rosu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010)

28. Sewell, P., et al.: Ott: effective tool support for the working semanticist. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming ICFP 2007, pp. 1–12. ACM, New York (2007)
29. Stefanescu, A., Park, D., Yuwen, S., Li, Y., Rosu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, 30 October–4 November 2016, pp. 74–91 (2016). https://doi.org/10.1145/2983990.2984027
30. Vacchi, E., Cazzola, W.: Neverlang: a framework for feature-oriented language development. Comput. Lang. Syst. Struct. **43**, 1–40 (2015)
31. Ward, M.P.: Language oriented programming. Softw.-Concepts Tools **15**, 147–161 (1995)