Extending the FSyntax/Hiord Approach with Imperative Notation*

Paula Corral[1,2], Jose F. Morales[1,2],
Pedro Lopez-Garcia[2,3], and Manuel V. Hermenegildo[1,2]

[1] Universidad Politécnica de Madrid (UPM)
[2] IMDEA Software Institute
[3] Spanish Council for Scientific Research (CSIC)
{paula.corral,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

Abstract. State variables, loops, and other features of imperative programming languages can bring coding simplification for certain programming idioms that are more cumbersome to express recursively. Because of this, some logic programming systems have incorporated various imperative constructs. `FSyntax` is a syntactic approach to supporting functional notation in `Prolog` systems which is based on the use of the syntax and term expansion facilities of the language. `Hiord` is also a syntactic approach to supporting higher-order in `Prolog`, building on `call/n`, and adding other features such as anonymous predicates. Both are used extensively, for example, in the `Ciao Prolog` system. In this paper, we propose a number of imperative-style constructs based on extending `FSyntax` and `Hiord`. These extensions are designed to combine nicely with the basic functional notation and the higher-order facilities, as well as with other extensions, such as constraints. In contrast to other proposals, our approach provides a set of primitives and a higher-level mechanism that, together, allow users to easily extend the language with features such as array notation, state variables, loops, etc. We illustrate the approach by defining a set of such features and using them to translate idiomatically in imperative style a large collection of small but interesting programs from the Euler Project, for which imperative-style implementations are available in multiple languages. We also show that the approach offers competitive performance.

Keywords: Imperative Constructs in Declarative Languages; Syntactic Extensions; Logic and Functional Programming; Higher Order; Prolog.

1 Introduction

Declarative programming allows for the efficient development of complex software systems while also helping in achieving correctness and safety. However, for

* Partially funded by MICIU projects CEX2024-001471-M *María de Maeztu*, and TED2021-132464B-I00 *PRODIGY*, as well as by the Tezos foundation. We also thank J. Fruhman, H. Kjellerstrand, W.W. Rong, B. Zhou, and N.F. Zhou for making their code for the Euler programs available. Finally, we would like to thank the anonymous reviewers for their very valuable and constructive feedback.

```
:- module(_, _, [functional]).                                    run ▶

primes(Limit) := ~sift(~integers(2, Limit)).

integers(Low, High) := ( Low =< High ? [Low | ~integers(Low+1, High)] | [] ).

sift([])     := [].
sift([I|Is]) := [I | ~sift( ~remove(Is, I)) ].

remove([], _)     := [].
remove([I|Is], P) := ( I mod P =\= 0 ? [I | ~remove(Is, P)] | ~remove(Is, P) ).
```

Fig. 1: Classical declarative example for computing primes in Prolog + `FSyntax`.

certain problems, it can sometimes be easier or more convenient for the programmer to use syntactic constructs and features borrowed from other programming paradigms. For example, `FSyntax` [5] is a syntactic approach to supporting functional notation in `Prolog` systems which is based on the use of the syntax and term expansion facilities of the language. `Hiord` [3] is also a syntactic approach to supporting higher-order in `Prolog`, building on `call/n`. Together they bring in the syntactic convenience of functional and higher-order constructs and both are used extensively in the `Ciao Prolog` system [10]. In this paper our focus is instead (or, more precisely, in addition to) on convenient syntactic constructs from imperative programming. For example, state variables and loops are important features of imperative programming languages that can bring coding simplification for certain algorithms that are more cumbersome to express using recursion and single assignment. Also, descriptions of algorithms in papers and textbooks are most often presented using imperative pseudocode, containing loops and other imperative constructs. Such algorithms can be implemented in logic languages using recursion, frequently rather elegantly, but this encoding can also in some cases be awkward and/or blur the correspondence with the original algorithm description.

Consider, for example, the *Sieve of Eratosthenes* algorithm for computing primes (we will refer to it simply as Eratosthenes' algorithm). It is a classic example in both functional and logic programming, where it is typically presented in an elegant recursive form. Figure 1 shows a version of this classic encoding, using the `FSyntax` package in Ciao Prolog. [4] Note that `FSyntax` allows tilde-annotated predicates to be written in function application style, e.g., `Z = ~p(~sort(X))` is expanded to `sort(X,Y), p(Y,Z)`.

This program is relatively simple, and arguably illustrates the elegance of declarative programming. Unfortunately, this classic encoding does *not* implement Eratosthenes' algorithm, but rather a naive algorithm known as *trial division* (see O'Neill [14]). If we want to implement the actual algorithm by Eratosthenes, we can perhaps turn to the Wikipedia, which provides a description of the algorithm in imperative pseudocode, shown in Figure 2. This algorithm

[4] This is the classical eager version. For completeness, a lazy version (from [5]) is shown in Figure 7, which is essentially the same as in lazy functional languages.

```
input: integer N > 1
output: list of prime numbers from 2 to N

A = array with index from 2 to N with all the values set to True

for i = 2,3, ..., sqrt(N) do
    if A[i] == True
        for j = i*i, i*i + i, i*i + 2i, ... below or equal to N do
            A[j] = False

return list with all k where A[k] is True
```

Fig. 2: Eratosthenes' algorithm in pseudocode (based on Wikipedia).

```
:- module(_, _).
:- use_module(library(logarrays)).

primes(N,Res) :-
    new_array(A), % Initialize an extendable array
    To is floor(sqrt(N)), % Just need to go to sq root of n
    complete_sieve(2,To,N,A,CompleteSieve), % Complete the sieve
    % Create a list with the primes
    take_primes(2,N,CompleteSieve,Res).

complete_sieve(Curr,To,_N,Sieve,RSieve):-
    Curr > To, !, RSieve = Sieve.
complete_sieve(Curr,To,N,Sieve,RSieve) :-  % If it is marked (0) it is not prime
    aref(Curr,Sieve,_El), !,
    NewCurr is Curr + 1,
    complete_sieve(NewCurr,To,N,Sieve,RSieve).
complete_sieve(Curr,To,N,Sieve,RSieve) :-  % Gets here if it is not marked (0)
    From is Curr * Curr,
    set_multiples(From,Curr,N,Sieve,Sieve1), % Mark with 0 multiples of a prime
    NewCurr is Curr + 1,
    complete_sieve(NewCurr,To,N,Sieve1,RSieve).

set_multiples(Curr,_Step,To,Sieve,RSieve) :-
    Curr > To, !, RSieve = Sieve.
set_multiples(Curr,Step,To,Sieve,RSieve) :-
    aset(Curr,Sieve,0,Sieve1),
    NewCurr is Curr + Step,
    set_multiples(NewCurr,Step,To,Sieve1,RSieve).

take_primes(Curr,N,_Sieve,Res) :-
    Curr > N, !, Res = [].
take_primes(Curr,N,Sieve,Res) :- % If it is marked (0) it is not prime
    aref(Curr,Sieve,_El), !,
    NewCurr is Curr + 1,
    take_primes(NewCurr,N,Sieve,Res).
take_primes(Curr,N,Sieve,Res) :- % Not marked (0): add it to Res
    Res = [Curr| Rest],
    NewCurr is Curr + 1,
    take_primes(NewCurr,N,Sieve,Rest).
```

Fig. 3: Eratosthenes, direct Prolog coding, using expandable, logarithmic arrays.

can be coded in standard Prolog (see, e.g., Figure 3), but the result is not very compact and can feel a bit awkward. A version coded using again `FSyntax` is shown in Figure 4. In these two cases we have used the `logarrays` library of expandable, logarithmic arrays. Other array implementations provide different performance/flexibility trade-offs. For example, a version using (arbitrary arity)

```
:- module(_, _, [functional]).                                          run ▶
:- use_module(library(logarrays)).

primes(N) := Res :-
    complete_sieve(2,~floor(sqrt(N)),N,~new_array,CompleteSieve),
    take_primes(2,N,CompleteSieve,Res).

complete_sieve(Curr,To,N,Sieve) :=
    ( Curr > To              ? Sieve
    | aref(Curr,Sieve,_El) ? ~complete_sieve(Curr+1,To,N,Sieve)
    | ~complete_sieve(Curr+1,To,N,~set_multiples(Curr*Curr,Curr,N,Sieve)) ).

set_multiples(Curr,Step,To,Sieve) :=
    ( Curr > To ?  Sieve
    | aset(Curr,Sieve,0,Sieve1) ? ~set_multiples(Curr+Step,Step,To,Sieve1) ).

take_primes(Curr,N,Sieve) :=
    ( Curr > N             ? []
    | aref(Curr,Sieve,_El) ? ~take_primes(Curr+1,N,Sieve)
    | [Curr| ~take_primes(Curr+1,N,Sieve)] ).
```

Fig. 4: Eratosthenes' algorithm in Prolog + `FSyntax`.

```
:- module(_, _, [functional]).                                          run ▶

primes(N) := Res :-
    A = ~functor(~,a,N),
    complete_sieve(2,~floor(sqrt(N)),N,A),
    take_primes(2,N,A,Res).

complete_sieve(Curr,To,N,Sieve) :-
    Curr > To -> true
    ; arg(Curr,Sieve,El), nonvar(El) -> complete_sieve(Curr+1,To,N,Sieve)
    ; set_multiples(Curr*Curr,Curr,N,Sieve), complete_sieve(Curr+1,To,N,Sieve).

set_multiples(Curr,Step,To,Sieve) :-
    ( Curr > To -> true
    ; arg(Curr,Sieve,0), set_multiples(Curr+Step,Step,To,Sieve) ).

take_primes(Curr,N,Sieve) :=
    ( Curr > N                        ? []
    | arg(Curr,Sieve,El), nonvar(El) ? ~take_primes(Curr+1,N,Sieve)
    |                                   [Curr | ~take_primes(Curr+1,N,Sieve)] ).
```

Fig. 5: Eratosthenes' algorithm in Prolog, using `FSyntax`, and terms for arrays.

terms and `arg/3` for the arrays is shown in Figure 5. The performance of this version, in the default bytecode grade, is comparable to that of the equivalent Python version, shown in Figure 6.

The versions of Figures4 and 5 are arguably more compact and elegant than that of Figure 3, but they still suffer from some of the previously mentioned issues. In particular, the correspondence with the original pseudocode is not obvious, at least to the untrained eye.[5] This brings us back to the idea that there are cases where incorporating syntactic features from imperative programming can bring programmer convenience and in general potentially also contribute to

[5] Note that coding the actual Eratosthenes algorithm in a lazy functional language is not trivial, as shown by O'Neill [14], and suffers from the same issues.

```python
import math

def primes(n):
    # Initialize the array with n+1 values to access until index n
    a = [True] * (n+1)

    # The loop goes until the square root of n
    to = math.floor(math.sqrt(n))

    # We cross out the non primes
    for i in range(2,to+1):
        if a[i]: # If a[i] is prime we cross out its multiples
            for j in range(i*i,n+1,i):
                a[j] = False

    # We take the primes from the sieve and put them into a list
    result = []
    for k in range(2,n+1):
        if a[k]:
            result.append(k)

    return result
```

Fig. 6: Eratosthenes' algorithm, in Python.

```
:- module(_, _, [fsyntax, lazy]).                                    run ▶

:- use_module(library(lazy/lazy_lib), [take/3, nums_from/2]).

:- lazy fun_eval cut/1.
cut([])      := [].
cut([H | T]) := [H | ~cut(~cut_(T, H))].

:- lazy fun_eval cut_/2.
cut_([], _)        := [].
cut_([H2 | T], H1) := R :-
    R = ( H2 mod H1 > 0 ? [H2 | ~cut_(T, H1)] | ~cut_(T, H1) ).

:- lazy fun_eval primes/0.
primes := ~cut(~nums_from(2)).

test_primes(N) := ~take(N, ~primes).
```

Fig. 7: Classical declarative example for computing primes (trial division): *lazy* version, using `FSyntax`. Note that `N` here is the *number* of primes computed.

the wider adoption of declarative languages. Also, depending on their semantics, the use of loops and other imperative constructs can sometimes help static analyzers by providing implicit information such as determinacy, non-failure, modes, types, and bounds on the number of times the loops are executed, which can be useful, e.g., for cost and complexity analysis.

Motivated by this, and building on the `FSyntax` and `Hiord` functional notation and higher-order facilities, we develop a set of primitives and a higher-level mechanism that together allow users to easily and selectively extend the language with imperative features. This generalization of `FSyntax`, which we call `xsyntax`, also allows more fine-grained control of the different extension components, so

that they can be activated and deactivated selectively at a finer granularity level: e.g., being able to use state variables without enabling functional notation. We also propose within `xsyntax` a `notation` facility, a convenience built over the expansion mechanisms of [2] that allows easily defining replacements for given term patterns.

Related work: Regarding the declarative representation of state, Definite Clause Grammars (DCGs), can be used to represent, thanks to their implicit arguments, both the previous and next state of a variable. DCGs can be extended to track multiple variables by encapsulating them in a single structure, but this can be difficult to manage. To address this, extended DCGs were introduced by Peter Van Roy [15], which allow simultaneous updates to multiple variables. However, EDGCs are still a comparatively less intuitive method for state management. A more direct approach is offered by languages like `Picat` [21,22], which uses the `:=` operator for updating state variables. Mercury uses the `!` notation to denote state variables and automatically expands variables into two to represent the state before and after a state change [9]. In addition to state variables, logic programming systems often support some form of mutable variables and/or the `setarg/3` primitive, which allows destructive updates to term arguments. This is typically used for localized updates to large data structures—such as array assignments– rather than for tracking the state of individual variables. However, this kind of destructive assignment is at odds with the declarative nature of logic programming, as it complicates the semantics significantly. Also monads, originally from functional programming, offer a structured way to manage state changes while preserving declarative semantics, somewhat related to DCGs. For example the functional-logic language Curry [8] uses monads to handle effects like I/O. The addition of monadic abstractions to Prolog via higher-order extensions and syntactic transformations has received some attention [12]. However, monads are arguably a less natural solution in logic programming.

A number of multiparadigm programming languages exist that integrate features from logic-, functional-, constraint-, or imperative programming. Alma-0 [1] was one of the first approaches, where imperativeness (as an extension of Modula-2) played a very prominent role, whereas newer proposals start as us from a declarative programming foundation. Some of these proposals are entirely new languages, rather than extensions of Prolog, in contrast to our approach. For example, `Picat`, in addition to the previously mentioned state variables, also incorporates two types of loops and array access notation [22]. MiniZinc [13,18] is a popular modeling language for combinatorial problems. It provides a loop construct which can be used to place constraints that depend on the loop bounds in a natural and regular way.

Returning to approaches based on extending Prolog with new syntactic constructs, as in our proposal, apart from the already mentioned `Hiord` and `FSyntax` extensions in `Ciao Prolog`, for instance `ECLiPSe` introduces *logical loops* as a language extension [17], and array notation, although it does not include state variable support. SICStus [4] has traditionally supported `setarg/3` and later incorporated *mutables*. XSB [19] also supports `setarg/3` and has an array facil-

ity. SWI [20] has global variables (also supported by Yap [16]) and some other extensions. François Fages has developed a mathematical modeling library for Prolog [6], inspired by MiniZinc. This library brings constraint modeling capabilities to Prolog with a mathematical focus. It includes the forall loop notation, which, like MiniZinc, is limited to verifying that a constraint holds across all iterations and does not support state variables. To the best of our knowledge, none of the current Prolog-based systems integrate all of the features that we address herein or use the compositional approach proposed.

2 Core primitives and their translations

This section introduces and describes the primitives and translations used in our approach, without committing yet to a specific syntax, as the building blocks that later will be used to define specific notations in user programs. In order to combine imperative constructs with other expansions, our approach introduces a generalization of `FSyntax`, called `xsyntax` that coordinates the expansion of all the primitives above, and performs a controlled fine-grained expansion of notations. A design objective is to be able to use one extension, like state variables, without necessarily having to use another, like functional notation.

The following sections describe all the primitives and their translations. We begin with closures, anonymous predicates that *capture* variables from their environment; follow with state variables, abstractions of logical mutable updates using threaded variables; and finally address loop constructs, code that iterates while a condition is met.

2.1 Higher-Order and closures

The `Hiord` extension (`hiord` package) enables higher-order untyped logic programming, allowing the declaration of anonymous predicates and closures in term positions. A closure is simply a predicate that captures variables from its surrounding environment. As an instrumental part of this work, we have reworked and enhanced the `Hiord` closures in two ways, referring to this extended version as `hiordx`. In particular, we have extended the declaration of captured variables, so that, in addition to specifying which variables are shared (already possible in `Hiord`), we can also declare non-shared variables, as follows:

$$\texttt{P = \{[}\mathit{ShVars}\texttt{] -> ''(}A_1,\ldots,A_n\texttt{) :- }\mathit{Body}\texttt{\}} \tag{1}$$

$$\texttt{P = \{-[}\mathit{NonShVars}\texttt{] -> ''(}A_1,\ldots,A_n\texttt{) :- }\mathit{Body}\texttt{\}} \tag{2}$$

$$\texttt{P = \{''(}A_1,\ldots,A_n\texttt{) :- }\mathit{Body}\texttt{\}} \tag{3}$$

The closure syntax (1) specifies *positive* sharing, where none of the parent environment variables are captured, except those listed as shared variables (*ShVars*). In contrast, syntax (2) specifies *negative* sharing, where all parent environment variables are captured, except those listed as non-shared variables (*NonShVars*). Finally, syntax (3) specifies *default* sharing, that is equivalent to the negative sharing of all variables from the terms in head arguments, which

is a useful convention in practice. Other enhancements of `hiordx` with respect to `Hiord` concern performance. In particular, the translation generates auxiliary predicates as needed, rather than higher-order callable terms, so that no run-time overhead is incurred from declaring anonymous predicates.

Translation of closures: Syntactic closures, which can appear in arbitrary term positions (in curly braces), are first translated to an internal literal representation by `xsyntax`. [6] This internal representation is then handled by another translation step, which we will refer as `xcontrol`, to compute the actual shared variables required in the closure, resulting only in a positive sharing list. Note that although positive sharing can be easily implemented as a goal-local translation step, negative or default sharing requires keeping track of variable lifetimes. This can be complex in the presence of other closures or control structures. Our implementation reuses the same lifetime tracking algorithm that computes shared variables in if-then-elses, disjunctions, and negated goals in the compiler, which annotates for each variable if it appears in more than one scope.

We have also included two new optimizations beyond those in the original `Hiord` implementation:

- Once the translation process has left only positive sharing closures given by the literal (`P = {[`$ShVars$`] -> ''(`$A_1, \ldots, A_n$`) :-` $Body$`}`), these are expanded as `P = auxpred(`$ShVars$`)`, with `auxpred` a fresh different name, and a new clause '`auxpred(`$ShVars, A_1, \ldots, A_n$`) :-` $Body$.', [7], so that `P` can be called as usual with `call/n`
- In some cases, the (auxiliary) predicate referenced by a closure may be known statically when using `call/n`. In general, this requires global program analysis, but in certain cases (such as those for loops we will see) the propagation is quite simple. In those cases, the compiler can completely translate the code into static calls, without needing to create a closure term or perform a dynamic call.

The use of negative and default sharing closures to manage variable scopes will be essential to simplify the introduction of the loop control structures. Moreover, the optimization of closure compilation, in addition to having value on its own, also makes the implementation of efficient loop constructs easier.

2.2 State variables

State variables provide a means to model state transitions in a purely declarative fashion, representing and managing mutable state by explicitly threading state values through predicates as additional input and output arguments. As mentioned in the introduction, this concept is closely related to Definite Clause

[6] This also ensures that other expansions, such as functional notation, are performed at the correct step.

[7] For clarity of presentation, the actual translation shuffles arguments at call time (using a dedicated internal functor for closures) to preserve first-argument indexing, which is assumed for predictable performance in Ciao.

Grammars (DCGs) in Prolog and has been generalized as Extended DCGs to allow tracking multiple states. However, in this work we adopt a more convenient syntax based on the state variables of Mercury [9].

We now briefly introduce the syntax and semantics of such state variables. Given a state variable (`S`) passed as an argument (denoted as `!S`, e.g., `p(!S)`), each predicate receives the *current version* of the variable as input (a logical variable) and returns an *updated version* as output (e.g., `p(`S_0`,`S_n`)`, where n is the last version number of the variable in the clause body). Similarly, each body literal can refer to the current version of the state variable (e.g., `p(S)` as `p(`S_k`)`, where k is the current version of the variable), or call a predicate that performs an update (e.g., `p(!S)` as `p(`S_k`, `S_{k+1}`)`). The sequence of variable versions represents the sequence of updates for that variable. Usually an assignment operator is provided as a special literal that just performs an update (e.g., `S := V` as S_{k+1} `= V`).

Similar to the treatment of closures, the user-level syntax for state variables is delegated to the `xsyntax` package, which is customized to transform high-level constructs into a set of intermediate primitives. These primitives are processed later by the `xcontrol` translation phase, responsible for managing versions of state variable across control structures.

Syntactic lowering: As mentioned before, this first phase is performed by `xsyntax`, which specializes in expanding terms to give them a meaning. It defines the syntax for representing and modifying state and performs the following transformations:

- The syntax `!X` as an argument of a structure is expanded as a pair of arguments $\texttt{X}_\circ$`, `$\texttt{X}_\bullet$ (read as "before X" and "after X") that represents the initial version and updated versions of the state variable, respectively.
- The goal `X :=V`, denoting an imperative assignment of state variable `X` to value `V` (which itself can be a logical variable), is translated into the form $\texttt{X}_\bullet$ `= V`. That is, the unification of the update version of `X` with `V`.

Translation of state variables: The translation of state variables is performed as part of the `xcontrol` phase. We introduce some notation to describe this process. Let Γ be a state variable environment mapping each state variable to a (logical) variable representing its current state; $\Gamma(S)$ obtains the value (logical variable) for S in the environment; $\Gamma[S \mapsto V]$ obtains a new environment where the value for S is updated to V. The environment is initialized with all the state variables in the clause (including variables in the head and other state variables appearing in the body). We define a translation for terms and goals $\mathsf{T}[\![t]\!](\Gamma) = (t', \Gamma')$, where t is a term or goal, t' is its translated form, Γ is the initial environment, and Γ' is the resulting environment. This translation is then applied to the clause body. Note that at this point other notations (such as functional notation) have already been expanded by `xsyntax`.

The translation of terms or literals (except conjunctions and disjunctions and other control structures) is as follows:

$$\begin{aligned}&\mathsf{T}[\![t]\!](\Gamma) = (t', \Gamma') \quad \text{where:}\\ &\quad \Gamma' = \Gamma[X \mapsto \textit{new-fresh-var} \mid X_\bullet \text{ occurs in } t]\\ &\quad t' = t \text{ with all } X_\circ \text{ replaced by } \Gamma(X), \quad \text{and all } X_\bullet \text{ replaced by } \Gamma'(X)\end{aligned}$$

The translation of control structures is as follows. Conjunctions are translated by sequentially threading the environments:

$$\begin{aligned}\mathsf{T}[\![G_1,\ G_2]\!](\Gamma) &= ((G_1',\ G_2'),\ \Gamma_2)\\ \text{where } (G_1', \Gamma_1) &= \mathsf{T}[\![G_1]\!](\Gamma),\\ (G_2', \Gamma_2) &= \mathsf{T}[\![G_2]\!](\Gamma_1)\end{aligned}$$

Treatment of disjunctions is more involved. The translation processes each branch individually, applying updates based on the environment at the disjunction point. After translating both branches, their resulting environments are unified. Specifically, if a state variable is updated in both branches, it unifies the two resulting states. If the state variable is updated in only one branch, it adds a new clause to the other branch that unifies the current state of the first branch with the environment at the disjunction point:

$$\begin{aligned}\mathsf{T}[\![G_1\ ;\ G_2]\!](\Gamma_0) &= ((G_1', U\ ;\ G_2', U),\ \Gamma')\\ \text{where } (G_1', \Gamma_1) &= \mathsf{T}[\![G_1]\!](\Gamma_0),\\ (G_2', \Gamma_2) &= \mathsf{T}[\![G_2]\!](\Gamma_0),\\ (U, \Gamma') &= \mathsf{Join}(\Gamma_1, \Gamma_2, \Gamma_0)\end{aligned}$$

The join operator $\mathsf{Join}(\Gamma_1, \Gamma_2, \Gamma_0) = (U, \Gamma')$ is defined as follows. For each variable X, $\Gamma_i(X) = x_i$, and:

- If $x_1 = x_2$: set $\Gamma'(X) = x_1$ and emit no unification.
- If $x_1 \neq x_2$: introduce fresh x', set $\Gamma'(X) = x'$, and emit unifications $x' = x_1$, $x' = x_2$.
- U is the conjunction of all such unifications.

2.3 Loops

In declarative languages, loops are not primitive constructs but must be encoded using recursion. The purpose of loop syntax is to provide a more natural and expressive way of modeling iteration, often involving mutable state variables that evolve across iterations. In this section we define a general-purpose logical loop construct as a building block (purely internal, and not exposed to the user), without committing to any fixed concrete syntax. As for previous primitives, the concrete user-level syntax will be handled by an `xsyntax` expansion, but for the case of loops, this will be described in more detail in Section 3.

Primitive loop construct: A loop `'$loop'(Vars, Init, Cond, Goal)` consists of the following components:

- A set of *iteration variables*, scoped to each iteration.
- An *initialization goal*, evaluated once before the loop starts.
- A *condition*, tested at the beginning of each iteration.
- A *body goal*, executed if the condition holds, also must prepare state for the next iteration.

Operationally, the loop executes the initialization first, and then repeatedly: it checks the condition, and if true, executes the body, prepares the state for the next iteration, and loops again. When the condition fails, the loop terminates. This corresponds to the recursive schema:

«begin» `:-` *Init*, *«body»*.
«body» `:-` (*Cond* `->` *Goal*, *«body»* `;` *true*).

Note that the use of "->" in the schema implicitly introduces a *cut*. We additionally support pairs of conditions with the syntax *posneg(PosCond, NegCond)*, that is then expanded into the following pure recursive schema.

«begin» `:-` *Init*, *«body»*.
«body» `:-` (*NegCond* `;` *PosCond*, *Goal*, *«body»*).

Note that such a recursive schema without cut is more convenient for some use cases, such as when using breadth-first and other search strategies, program analyses or transformations, running "backwards" (supporting several modes), etc.

Translation of loops: Translation of loops requires identifying all the state variables involved during iterations. This is performed in the `xcontrol` step as part of the variable lifetime analysis. Loop constructs are annotated as:

`$shloop(Vars, StLoopVars, Init, Cond, Goal)`

where `StLoopVars` is the set of state variables (S_1, ..., S_n) potentially updated within the loop.

The final loop form is finally translated to nested closures, where each closure introduces its own scope and versioned state variables. These closures are then statically compiled into plain predicates (similary for the pure recursive schema):

```
Begin = {
  ''(!S1, ..., !Sn) :-
    Init,
    Body = {
      -[Vars] -> ''(!S1, ..., !Sn) :-
        ( Cond -> Goal, Body(!S1, ..., !Sn) ; true )
    },
    Body(!S1, ..., !Sn)
},
Begin(!S1, ..., !Sn).
```

Each closure introduces new versions of the state variables through the transformation rules described in Section 2.2. Since each iteration forms a distinct scope, proper versioning and unification of state is required to maintain correctness. The use of negative-sharing in *Body* ensures that the marked iteration variables are independent through recursive invocations.

Once transformed into this canonical form, the `hiordx` translation (see Section 2.1) eliminates the closures via specialization, resulting in efficient (as hand-written recursions), statically compiled code that faithfully implements the loop semantics in a declarative setting.

3 Defining a concrete imperative syntax

As mentioned before, in our proposal, we make an explicit separation between the higher-level syntactic constructs and the core primitives (such as closures, state variables, and loops) necessary to define such constructs in a coherent and composable way that fits in a logic programming setting. Following this idea, this section presents an example concrete proposal for imperative syntax, built on top of the previous primitives, and the `xsyntax` expansions.

More flexible notation: `FSyntax` allows customization of expansions using the `fun_eval` declaration. For example, `:- fun_eval arith(true)` enables the evaluation of arithmetic functors (in a module or module context) as Prolog arithmetic functions (`p(A+B)` expands to `X is A+B, p(X)`), or `:- fun_eval append/3` enables functional expansion of, e.g, `p(append([1],[2]))` to `append([1],[2],X), p(X)` without having to explicitly annotate the expansion (i.e., without the ~ in `p(~append([1],[2]))`). In `xsyntax` we offer a richer customization. First, it is possible to select different arithmetic evaluations. For example, `:- fun_eval arith(clpfd)` expands `p(A+B)` as `X #= A+B, p(X)`, which is convenient for constraint modeling, running imperative code backwards, etc. Another notable addition is the possibility of defining notation patterns with `:- notation(Pattern, T)`, where all occurrences that match `Pattern` are replaced by `T` (the "notation" name is intended to evoque the process of introducing notation in mathematical text). For example, `:- notation(X ∈ Ls, member(X,Ls))` expands `X ∈ [1,2,3]` to `member(X, [1,2,3])`. Note that these declarations are just user-definable notational aids which do not change the underlying semantics of Prolog. They are a user-friendly convenience built over the more traditional (and also more powerful) term expansion mechanisms –in particular, those of [2]. This notation facility will be useful in what follows to easily map the higher-level imperative constructs to the lower level primitives.

Array notation: Arrays are a fundamental data structure in programming, especially in imperative languages. They also offer a natural way to represent mathematical objects such as vectors, matrices, or functions over finite domains, which are useful in modeling constraints or mathematical problems. We use the previously introduced notation declarations to define custom syntax for accessing

array elements and to overload the assignment operator for element updates, analogous to function update in mathematical logic: [8]

```
:- notation(Arr[I], ~get_elem(Arr,I)).
:- notation(Arr[I]:=Val, (Arr := ~replace_elem(Arr,I,Val))).
```

The notation above maps access and updates to `get_elem/3` and `replace_elem/4` predicates. If defined as *multifile* (e.g., as described in [7]), these predicates can act as a bridge interface between several array-like data structures. To test this we implement a package for array-like syntax and interface (including other useful operations like `array_length/2`). We provide this unified interface with non-destructive arrays with logarithmic access (`library(array_log)`), functor-based arrays with O(n) update operations, destructive mutable arrays using `setarg/3`, as well as linked lists with nth-element accessor and update operations.

While loops: We can define a *while* loop simply by setting an empty initialization and iterating *Goal* while the condition *Cond* holds:

```
:- notation(while (Cond) { Goal },
            '$loop'([], true, Cond, Goal)).
```

For loops with iterators: Similarly to *while* loops, it is possible to define *for* loops that iterate over elements extracted from an *iterable* object. As with arrays, we define a simple interface to iterable objects (terms) as *multifile* predicates, as follows:

- `iter_cond(Curr,X)`: obtains the iterator value `X` for the current state `Curr`; fails if there are no more values.
- `iter_next(Curr, Curr2)`: transitions from `Curr` to next state `Curr2`.

For example, we can define iterators over lists or integer ranges as follows:

```
iter_cond([X|_], X).
iter_next([_|Xs], Xs).

iter_cond(range_iter(B, _Step, X), X) :- X=<B.
iter_next(range_iter(B,Step,Curr), range_iter(B,Step,Curr2)) :-
     Curr2 is Curr+Step.
```

We can then use a notation declaration to define a *for* loop over an iterable as follows:

```
:- notation(for (I in Iter) { Goal },
            '$loop'([I],
                    Curr := Iter,
                    iter_cond(Curr, I),
                    (Goal, Curr := ~iter_next(Curr)))).
```

[8] Note that the same approach can be used to offer notation for access and update of other indexed data structures like dictionaries.

```
:- module(_, _, [functional,loops,arrays]).

:- use_module(library(arrays/arrays_log)).

primes(N) := Res :-
    % Initialize the array with n+1 values to access until index n
    A = ~new_array_log,
    for (I in 2 .. N) { A[I] := true },
    % We cross out the non primes
    for (I in 2 .. ~floor(sqrt(N))) { % Loop goes until square root of N
        if (A[I] == true) { % If a[i] is prime we cross out its multiples
            for (J in I*I .. I .. N) { A[J] := false }
        }
    },
    % We take the primes from the sieve and put them into a list
    Res = ResTail,
    for (K in 2 .. N) {
        if (A[K] == true) { accum(!ResTail,K) }
    },
    ResTail = [].

accum(!R, X) :- R = [X|Tail], R := Tail.
```

Fig. 8: Eratosthenes algorithm, in Prolog + the developed syntax.

This represents generic code that would work for any data that implements the iterator interface. In particular, it is also convenient to define shorter syntax for some iterators (or alternatively, just map `Begin..End` as a notation for `range_iter(Begin,1,End)`):

```
:- notation(for (I in Begin..End) { Goal },
            for (I in range_iter(End, 1, Begin)) { Goal }).
```

Additionally, for efficiency, unfolded iterators can be provided using auxiliary notations that factorize definitions (note that the same effect can also be achieved through partial evaluation). As an example, this is the result for this range iteration:

```
:- notation(for (I in Begin..End) { Goal },
            '$loop'([I],
                    Curr := Begin,
                    Curr =< End,
                    (Goal, Curr := Curr + 1))).
```

Back to Eratosthenes: Figure 8 presents an encoding of the Sieve of Eratosthenes algorithm using a number of the extensions developed. It is relatively easy now to see the correspondance with the pseudocode in Figure 2, and with the Python version (Figure 6). Also, as mentioned before, the generated code is equivalent to the plain Prolog recursive versions, and the default bytecode grade compilation of Figure 8 is comparable in performance to the Python code.

4 Some experimental results

The proposed extensions are provided as separate Ciao packages `hiordx`, `loops`, `statevars`, and `arrays`, which can be selectively enabled as required. These

extensions are implemented as previously described and depend on the `xsyntax` extension and the `xcontrol` internal phases, which together provide coherent support for all the proposed features.

In order to test the convenience, implementation, and performance of these extensions, we have chosen to use a large set of problems from the Euler Project [11]. The choice of these benchmarks is motivated in part by the variety of problems and algorithms involved, which allows illustrating the expressiveness and flexibility of the proposed extensions, and also because encodings of many of these problems are available in several imperative languages, and in particular a good number of them are accessible from the `Picat` web site. As mentioned before, `Picat` is also a logic-based language, different from Prolog, that has loops, state variables, and array index notation in addition to constraints and tabling, and thus we consider it a very good point of reference.

We have encoded the examples in Ciao using our extensions. The resulting code[9] constitutes perhaps the best illustration of the approach. In particular, these programs test the use of the array-like structures, state variables, and loops presented, as well as other Ciao extensions like *clpfd*, *tabling*, *assoc*, or *pmrules*. The coding used typically involves an imperative syntactic style, but often makes use also of Prolog's search, unification, constraint solving, etc., thus illustrating how our approach allows synergistically mixing both styles.

Regarding performance, we have carried out a comparison with `Picat`, which is known to be quite performant, so we also consider it a very good point of reference from this point of view. In this sense our objective is not to perform an in-depth performance comparison, since both the implementation and the problem encodings can still be improved, but rather to have an estimation of whether the approach is competitive.

Table 1 shows the execution times of each program, in seconds, in `Picat` and `Ciao`, grouped in different subtables by the extensions or utilities they use. The symbol '`-`' means that no output/answer is produced for some reason. The experiments were run on a MacBook Pro, 3,1 GHz Dual-Core Intel Core i5; 16 GB 2133 MHz LPDDR3; macOS: Ventura 13.7.4 (22H420). [10] The results computed in both languages are identical for all cases.

Table 1a presents examples that use only loops, including state variables, without any other extensions. These examples demonstrate that Ciao's implementation of state variables and loops is competitive with `Picat`'s. We observe some performance differences, which may be due to variations in the virtual machine implementations. Table 1b presents examples that use tabling, showing similar performance overall. However, in `Picat`, they execute slightly faster and comparing this with Table 1a, we can infer that the difference is likely to be because of variations in the tabling implementation. Table 1c presents examples that use loops and index notation over lists from the new array extension. The performance is generally similar in both languages, with some exceptions such as

[9] See: `https://gitlab.software.imdea.org/ciao-lang/ciaoimp-benchmarks`

[10] In order to save space we have left out of the table the problems for which the execution times were too low to be significant.

File	Picat	Ciao	Picat/ Ciao
p004	0.750	0.102	7.288
p007	0.299	0.172	1.731
p012	11.250	6.975	1.612
p016	0.001	0.000	6.134
p020	0.001	0.000	13.157
p022	0.021	0.021	0.957
p025	3.392	2.117	1.601
p029	2.353	2.288	1.028
p030	1.114	2.220	0.501
p034	0.403	0.289	1.392
p036	1.250	0.841	1.485
p045	0.043	0.030	1.388
p046	0.041	0.027	1.517
p048	0.826	0.017	47.906
p056	0.781	0.210	3.704

(a) Examples that just use loops.

File	Picat	Ciao	Picat/ Ciao
p021	0.173	0.137	1.257
p027	1.823	4.482	0.406
p037	3.029	4.278	0.707
p041	0.012	0.014	0.834
p053	0.041	0.036	1.112
p055	0.073	0.070	1.029

(b) Examples that use loops and tabling.

File	Picat	Ciao	Picat/ Ciao
p017	0.011	0.016	0.665
p019	0.051	0.050	1.016
p024	1.150	8.596	0.133
p026	0.074	1.946	0.038
p040	0.197	0.161	1.218
p042	0.025	0.030	0.829
p060	142.493	91.048	1.565
p076	0.003	0.057	0.052
p077	0.013	0.204	0.063

(c) Examples that use loops and index notation in lists.

File	Picat	Ciao	Picat/Ciao
p010_log	-	9.778	
p010_mut	0.909	0.933	0.973
p047_log	-	10.553	
p047_mut	0.804	1.184	0.678
p050_log	0.282	1.909	0.147

(d) Examples that use loops and index notation in arrays.

File	Picat	Ciao	Picat/Ciao
Loops and assoc			
p032	2.398	2.005	1.195
p044	1.465	2.458	0.595
p062	0.032	0.168	0.190
Loops and pmrule			
p052	0.418	0.538	0.775
p049	0.138	0.113	1.218

(e) Examples that use loops and other extensions.

Table 1: Experimental results on Euler Project problems in Picat and Ciao.

p024, p026, p076, and p077. These are likely to be due to the fact that replacing an element in a list is destructive in `Picat` but not in Ciao. Table 1d presents examples that use loops and index notation over arrays from the new array extension. Specifically, they are implemented in Ciao using both logarithmic and mutable arrays, indicated by file name suffix. Performance in mutable arrays is similar to `Picat`'s, which makes sense since `Picat`'s array implementation is also mutable. Table 1e presents other examples that use loops. The examples using single-side unification rules (denoted as *pmrule*) show performance comparable with `Picat`. The other examples use *maps*, for which we have used simply the traditional `library(assoc)` module, and the performance difference is likely due to the faster destructive map updates in `Picat`.

In general, these results suggest that our implementation approach achieves the objective of supporting a customizable and rich set of imperative constructs within Prolog with competitive performance.

5 Conclusion

We have proposed a number of imperative-style constructs that build on and extend the `FSyntax` and `Hiord` syntactic extensions to Prolog. The proposed extensions have been designed so that they combine well with the basic functional notation and the higher-order facilities as well as with other extensions, such as constraints, tabling, etc. In addition, our approach is based on a set of primitives and a simplified, higher-level expansion mechanism that together have allowed us to easily add features such as array notation, state variables, loops, etc. We have also made on the way instrumental extensions to the previous work on `Hiord` and `FSyntax`. We have implemented and evaluated the proposed mechanisms by defining a set of imperative features and exercising their usefulness by translating idiomatically, in imperative style, but also using simultaneously Prolog's characteristics, a large collection of small but interesting programs from the Euler Project. Apart from their intrinsic interest, the choice of these benchmarks was also motivated by the fact that encodings of many of these problems are available in a number of imperative languages, and in particular in `Picat`, which, as we have argued, is a very good point of reference. We have also studied the performance of the translated programs. While some imperative-style constructs were previously available in some form or another in some Prolog systems, and more comprehensively in non-Prolog systems like `Picat`, we argue that our Prolog-based proposal is comprehensive, coherent, and extensible, as well as offering competitive performance.

References

1. Apt, K.R., Brunekreef, J., Partington, V., Schaerf, A.: Alma-O: An Imperative Language That Supports Declarative Programming. ACM Trans. Program. Lang. Syst. **20**(5), 1014–1066 (1998). https://doi.org/10.1145/293677.293679, https://doi.org/10.1145/293677.293679
2. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. pp. 131–148. No. 1861 in LNAI, Springer-Verlag (July 2000)
3. Cabeza, D., Hermenegildo, M., Lipton, J.: Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In: Ninth Asian Computing Science Conference (ASIAN'04). pp. 93–108. No. 3321 in LNCS, Springer-Verlag (December 2004)
4. Carlsson, M., Mildner, P.: SICStus Prolog – the First 25 Years. Theory and Practice of Logic Programming **12**(1-2), 35–66 (2012). https://doi.org/10.1017/S1471068411000482
5. Casas, A., Cabeza, D., Hermenegildo, M.: A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In: The 8th International Symposium on Functional and Logic Programming (FLOPS'06). pp. 142–162. Fuji Susono (Japan) (April 2006)
6. Fages, F.: A constraint-based mathematical modeling library in prolog with answer constraint semantics (2024), https://arxiv.org/abs/2402.17286

7. Garcia-Contreras, I., Morales, J., Hermenegildo, M.: Incremental Analysis of Logic Programs with Assertions and Open Predicates. In: Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). LNCS, vol. 12042, pp. 36–56. Springer (2020). https://doi.org/10.1007/978-3-030-45260-5_3
8. Hanus, M.: Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-language.org (2013), language Report
9. Henderson, F., Conway, T., Somogyi, Z., Jeffery, D., Schachte, P., Taylor, S., Speirs, C., Dowd, T., Becket, R., , Brown, M., Wang, P.: The Mercury Language Reference Manual. State Variables. The University of Melbourne (2014), https://mercurylang.org/information/doc-release/mercury_ref/State-variables.html
10. Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming **12**(1–2), 219–252 (January 2012). https://doi.org/10.1017/S1471068411000457, https://arxiv.org/abs/1102.5497
11. Hughes, C.: Project euler (2025), https://projecteuler.net
12. McGrail, R.: Monads and Control in Logic Programming. Ph.D. thesis, Wesleyan University (1999)
13. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming. pp. 529–543. Springer (2007)
14. O'Neill, M.: The genuine sieve of eratosthenes. J. Funct. Program. **19**, 95–106 (01 2009). https://doi.org/10.1017/S0956796808007004
15. Roy, P.V.: A Useful Extension to Prolog's Definite Clause Grammar Notation. ACM SIGPLAN Notices **24**(11), 132–134 (November 1989)
16. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming **12**(1-2), 5–34 (2012). https://doi.org/10.1017/S1471068411000512
17. Schimpf, J.: Logical loops. In: In Proceedings of the 18th International Conference on Logic Programming, ICLP 2002. pp. 224–238. SpringerVerlag (2002)
18. Stuckey, P.J., Marriott, K., Tack, G.: MiniZinc Documentation, https://docs.minizinc.dev/en/stable/index.html
19. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. Theory and Practice of Logic Programming **12**(1-2), 157–187 (Jan 2012). https://doi.org/10.1017/S1471068411000500
20. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494
21. Zhou, N.F.: Picat: A scalable logic-based language and system. In: 2nd Symposium on Languages, Applications and Technologies. Open Access Series in Informatics (OASIcs), vol. 29, pp. 5–6. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2013). https://doi.org/10.4230/OASIcs.SLATE.2013.5, https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.SLATE.2013.5
22. Zhou, N.F., Fruhman, J.: Picat Guide (2025), https://picat-lang.org/download/picat_guide.pdf