Jumble of Heuristic Notions

Hesam Samimi

Computer Science Department
University of California, Los Angeles
hesam@cs.ucla.edu

Abstract. We introduce an educational programming language with a syntax similar to a natural language. It is built around a theoretical programming model where the default mode of computation is declarative and non-deterministic, as opposed to imperative and highly optimized. Users can then specify declarative optimizations and heuristics that may essentially make the program imperative, without sacrificing compactness or clarity. We have written several applications in the language including a chess program and a register allocation component for an industry-level compiler. While these programs tend to be inefficient compared to highly optimized implementations, they are much more compact, easily understandable, and extensible. We believe this programming environment is suited for education of programming, and represents a model that may eventually become practical as multi-core hardware becomes mainstream.
Keywords: Programming Education, Programming Languages, Declarative Programming, Automated Planning, Artificial Intelligence

1 Introduction

We introduce a theoretical programming model called *Programming as Planning* (*PaP*) that aims to bring traditionally Artificial Intelligence notions such as search, heuristics, and planning into everyday programming. While PaP can be implemented as a library for existing languages, we designed a high level language that is natively built around this methodology. JOHN resembles a natural language and serves as an educational tool for children and new programmers. We named this language as a tribute to Artificial Intelligence and Programming Languages pioneer John McCarthy. [1]

1.1 Programming as Planning

Programming tasks can be viewed from the AI viewpoint of automated planning. Program specifications (declarative part) are considered *goals*, while the implementations (imperative part) are viewed as *actions* that try to realize those goals. The planning problem is optimizable by determining the right action (or

[1] Also an acronym for "Jumble of Heuristic Notions" by Alan Kay.

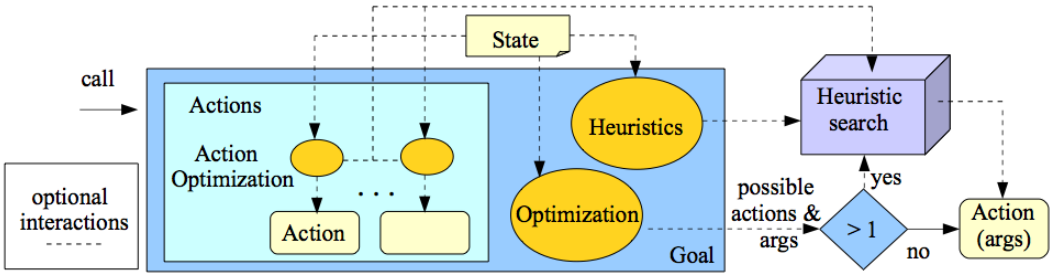

Fig. 1. Overview of *Programming as Planning*.

the set of plausible actions) to take at any given step in the execution of the program.

PaP adds non-determinism to the normal execution of an imperative program. At each instruction point, the code dynamically consults user-specified heuristics to determine the next instruction. It may also explore multiple possibilities by means of exhaustive search. The approach suffers from the overhead of extra computation to dynamically determine the next instruction at every step. On the other hand, it brings about the interesting possibility of non-deterministic computation by listing multiple actions to explore. *Optimizations* may be introduced to narrow down the set of possible actions to try at any given point, as well as *heuristics* that describe the preferred order in which to explore those possibilities. In this context, an imperative program is a fully optimized planning problem, where every action at every step of the program is predetermined.

An overview of PaP is shown in Figure 1. The solid lines represent a fully imperative program, where every call and the set of arguments is known. The optional dotted lines display the interactions between added optimizations and heuristics with the current state. These guide the program in the presense of non-deterministic choices and may cause the system to consult a heuristic search component to decide how to go forward.

We designed PaP to represent an idealistic programming framework where efficiency and practicality is traded in for a higher level of control and reasoning power. Our aim is to study models that we believe programming languages should soon move towards. We do not expect the methodology to be readily embraced by software developers due to its lack of focus on efficiency. On the other hand, we believe educating students should not be hindered by efficiency and legacy considerations, as unfortunately is the case today.

1.2 JOHN Language

JOHN is a light-weight object-oriented language which uses situation calculus [7], possible worlds models [2], and first-order logic to reason about the past and future states of the objects. The language supports the PaP methodology natively. Our main design objectives are:

-*Educational Tool for Children and New Programmers.* The syntax aims to read like natural language. We believe that the first thing programming students need to engage in should be thinking on how to model a domain specific problem in a natural way, as opposed to dealing with a steep learning curve for unfamiliar syntax and constructs.

-*Embedding AI into Programming.* We can add a higher level of control to programs if the procedures are not called directly, but a decision component chooses which combination of procedures to run and when. The program can use the aid of heuristics and optimizations to decide the most relevant set of procedures to run, as well as search for exploring multiple possibilities.

-*Separating Specifications and Implementations.* By adapting the goal-driven PaP model, programs become more coherent, readable, and manageable since they are constructed in a way that separates the specifications of problems (i.e. what it is they're trying to accomplish) from procedures and optimizations (i.e. how they accomplish that). Optimizations do not modify or clobber the program as they are kept separately and can easily be plugged in and out of the code.

2 Design

JOHN is a dynamically typed, message-passing style language that can be implemented on top of an existing object-oriented language. We use a non-standard terminology and use the term *microworld* for namespaces, *property* for instance variables, *action* for methods, and *goal* for boolean expressions that specify a desired state. `it` and `its` keywords are used interchangeably with the more common `this` or `self` keywords. The keyword `qualify` is used to denote a function (no side effects).

It turns out that many typical programming tasks can be done with a goal-oriented mind. The act of sorting a list for instance can be viewed as a `sorted` goal belonging to the `List` type, with potentially many candidate actions that may achieve it, such as merge or bubble sort algorithms. We will use this example to describe several features in the language.

Figure 2 defines a new type `AList` with an `items` property. We intend to use this property as a `List`, which is a primitive type. We use the `make` command to instantiate an object of this class named `list` that has an empty list `items` property. Types like `Integer` and `List` along with some basic operations are supported as primitives in the language. For example, the list primitive operations `+` and `-` append or remove an item, respectively. Next, in Figure 2 we define `add` and `delete` actions for the class to describe how change can come about in this microworld. The `add` action takes an integer argument `item`, and the `delete` action specifies a rule that it may not be applied to an empty list.

2.1 Worlds

Nothing is mutable in the JOHN language, and the properties of objects may not be explicitly modified. Object may change state only via actions. The state of

```
create AList items.
make AList list [ ].
action AList add: Integer item consequence its items = its items + item.
action AList delete consequence its items = its items - its items last.
rule AList delete its items empty not.
```

Fig. 2. Class definition, object instantiation, and action definitions

a microworld at any given time is called a *world* [1]. There is a *time* variable associated with each world. Every time an action is performed, the microworld moves into a new world of an incremented time that represents the new state for the microworld. The previous worlds are kept around.

The `time` command displays the time associated with the current world for the microworld. Time is initially 0 as Figure 3 shows for our previous `List` example. In Line 2, `list` performs the `add 1` action, which brings the microworld to time 1. Line 5 shows how the state of the microworld from past worlds can also be queried. If the set of all possible actions that a microworld can undergo at any given time is finite, then the system can reason about its possible future states. These are called *possible worlds* [2].

The `add` action in our example involves a primitive integer argument. Let's assume we bound the `Integer` type to 2 bits for the purpose of possible worlds reasoning. Querying an expression at a time distance n into the future implies the set of all possible values that the expression can take, after the execution of n arbitrary but valid actions. Consider again our example, where the microworld was at time 1 and the `items` list held the value `[ 1 ]`. The set of all possible actions for the current world is {`list delete, list add -2, list add -1, list add 0, list add 1`} and thus the possible worlds value for the `items` property at time 2 is as shown in Line 6 of Figure 3. Note that if other objects exist in the microworld then the actions by those should also be considered, and it is possible for an object property to retain its value in the new world. Line 7 shows the possible values for the list 2 time units ahead, which is a sizable set and not fully shown.

2.2 Goals

Possible-worlds reasoning is elegant as it leaves the current state unchanged. This is important since the microworld can do useful work in the meantime. Moving to a particular world is as simple as updating the current world pointer, and multiple worlds can easily be spawned, examined, and discarded. It also enables the system to use heuristic search to perform declarative computation. Let's now consider the task of sorting our list. We first need to specify the meaning of sorting. In Figure 4 we define the `sorted` function for the primitive type `List`. The figure demonstrates both the use of recursive definitions and first-order logic

```
1     > time.
      0
2     > list add 1.
      ok.
3     > list items.
      [ 1 ]
4     > time.
      1
5     > at time 0 list items.
      [ ]
6     > at time 2 list items.
      possible world values (1 time unit from now):
      { [ ], [ 1 -2 ], [ 1 -1 ], [ 1 0 ], [ 1 1 ] }
7     > at time 3 list items.
      possible world values (2 time unit from now):
      { [ 1 ], [ 1 -2 -2 ], [ 1 -2 -1 ], ... }
```

Fig. 3. Evaluating expressions in the past, present, and future worlds

```
qualify List sorted if its size < 2.
qualify List sorted if its first <= its second and its rest sorted.
qualify List permSubOf: List l if for all e in it |
    it count: e = l count: e.
qualify List permOf: List l if it permSubOf: l and l permSubOf: it.
goal AList sort its items sorted and
    its items permOf: at time 0 its items.
```

Fig. 4. Sort specification. `first`, etc. are primitive operations for this type.

expressions. The multiple definitions for `sorted` imply a disjunction in the order given.

Assuming the current world is at time 0, the goal predicate is specified in Figure 4. We can now ask the system to satisfy the sorting goal declaratively for the `list` object. This starts off a search of possible worlds, up to a limit time step into the future, for an action path that leads to a world where the goal is satisfied. But the only actions defined for our class are `add` and `delete` and thus the search in possible future worlds will time out with no success in sorting, as shown in Figure 5.

Once we have specified sorting, we can move onto adding implementations. To perform sorting as a planning problem, we implement sorting algorithms to make in-place updates and in such a way that each invocation runs through one iteration of the loop. Assuming we have already defined two more actions: `insertionSort` and `quickSort`, Figure 6 demonstrates how the `list` object can

```
> list satisfy sort.
no solution worlds within 10 time units.
```

Fig. 5. Satisfying goals declaratively using any available actions

```
> list items.
[ 2 7 1 5 ]
> list satisfy sort using insertionSort quickSort.
satisfied at time 3 by actions:
[ list insertionSort, list insertionSort, list insertionSort ]
```

Fig. 6. Satisfying goals non-deterministically.

now satisfy its sorting goal declaratively. The `using` keyword limits the set of actions explored explicitly.

2.3 Goal Optimizations

When satisfying goals we often have certain strategies and optimizations in mind that choose the best combination of procedures that achieve it. Similarly, procedures may require input arguments and themselves may allow optimizations for choosing the right value for those arguments.

Consider that we prefer to use `insertionSort` algorithm whenever a list is mostly sorted, but `quickSort` otherwise. This insight can be added to the program as a *goal optimization* (see Figure 7).

While this is surely a contrived example, it is well known that choosing the best sorting algorithm depends on the input. It is not difficult to find real world scenarios where better results are achieved if strategies are changed dynamically based on circumstances. JOHN allows addition of optimizations for goals, which has the effect of reducing possible worlds during heuristic search.

Figure 8 shows part of the description for the game of chess, where a `move` action is specified for the `Player` class. The action will have to specify rules for a valid move. Without optimizations, the next possible worlds are computed by trying all possible actions that the player can perform, namely 64 x 64 combination of squares to represent the origin and destination position for a specific move. This is safe, as only those argument combinations for which the action rule is valid will be actually explored. But this is inefficient because the rules are complex and time consuming to evaluate. We need to provide an optimization here to limit the set of arguments tried for making a move.

The first *action optimization* shown in the Figure 8 specifies that the set of arguments that should be tried during possible worlds generations should be statically pruned to a smaller set. This set will be determined by the `accessibleFrom`

```
qualify List nearlySorted if its unsortedness < 2.
optimization AList sort use insertionSort if its items nearlySorted.
optimization AList sort use quickSort if its items nearlySorted not.
```

Fig. 7. Optimizing Goals. `unsortedness` function description not shown.

```
action Player move: Square from Square to consequence ...
static optimization Player move to acessibleFrom: from.
dynamic optimization Player move from in its pieces position.
```

Fig. 8. Static and dynamic action optimizations.

function that filters out the pair of `from` and `to` arguments that cannot possibly constitute a chess move regardless of the state of the game (e.g. A1 to B4).

It is also useful to define dynamic optimizations to determine the right arguments for the action. The `Player` class has a `pieces` property to represent the set of pieces it has on the board, each of which has a `position` property. Unlike the previous static case, this set of piece positions depends on the current world. The last line in Figure 8 defines a dynamic action optimization that limits the `from` argument for the `move` action to be in this set. This will further reduce the number of invalid actions attempted during the possible worlds calculations.

2.4 Goal Heuristics

It is often the case that the possible worlds tree for some microworld is too large for a brute-force search. Search heuristics are used to effectively reorder the possible worlds generation by assigning a score to each possible world and exploring the highest scoring ones first.

In the presence of multiple heuristics, the cumulative score for all of them is considered. Heuristics are given weights to allow for prioritization. Figure 10 shows a few heuristics that one may consider for chess strategies. Here by providing a higher weight on the first two heuristics, we are teaching the computer that capturing pieces is more important that having pieces in strategic locations.

An arbitrary number of optimizations and heuristics such as these can be attached to goals and actions described in a microworld. With the aid of these optimizations and heuristics, objects can perform actions with one or all of the arguments missing. The system will non-deterministically choose one of the acceptable arguments specified by the optimizations and heuristics given to perform a desirable action (See Figure 9). These can be easily turned on and off in the code without modifying the rest of the program. This goes along with our intention to keep the optimizations in the program isolated, comprehensible, and extensible. Table 1 summarizes the optimizations allowed in the PaP model.

```
> player1 move.
ok. player1 performed move from E2 to E4.
```

Fig. 9. Running an action without arguments.

```
heuristic Player win 10 maximize its pieces worth sum.
heuristic Player win 10 minimize its opponent pieces worth sum.
heuristic Player win 1 maximize its centralizedPieces size.
```

Fig. 10. A few weighted heuristics for `win` goal in chess.

3 Implementation

We implemented JOHN as a stand-alone language on top of a Smalltalk implementation called COLA [9]. The documentation and download information are available on the project Wiki page at `http://www.cs.ucla.edu/~hesam/john`.

3.1 Implementing Worlds

Worlds are implemented as dictionary data structures in our prototype. As the state of the microworld changes, it spawns a new world as a child world. For efficiency, only the changing properties will be stored in the new world. Looking up a property at world of time t may have to go up the parent-world chain up to time *0* to find the property for a particular object. A full description of the world construct can be found in [1].

3.2 Search of Possible Worlds

A linear space, best-first search algorithm such as *Iterative Deepening A* (IDA*)* [6] is an appropriate search algorithm for our planner, as it guarantees finding optimal solutions without using too much memory. We use a variant of IDA* to add dynamic rules for node (possible worlds) generations. This will accommodate JOHN's goal and action optimizations. The dynamic rules of node generations and branch reordering enables us to optimize the planner for deterministic computation.

Another essential feature in our search implementation is the use of worlds as disposable states of the microworld at a particular point in time [1]. The planner should not interfere with the state of the microworld while inferring a solution, as the object may be doing other work. Committing to a given world is as simple as updating the current world pointer of the microworld to that world. The search algorithm is summarized in Figure 11.

Table 1. Summary of optimizations in JOHN

Type	Dynamic Optimization	Dynamic Heuristic
Effect	dynamic pruning of possible worlds	reordering visited possible worlds
Semantics	which actions to explore with what args	which worlds to explore first
Implementation	predicates attached to goals returning set of actions and predicates attached to actions constraining the args	numeric expression returning a score for a given world

```
IDAStar(goalName, currWorld):
  threshold := goalName_heuristic(currWorld).
  repeat:
    (1st, 2nd) := IDAStarIter(goalName, currWorld, 0, threshold).
    if 1st = true:
        return the 2nd which is the solution world.
    otherwise:
        threshold := 2nd which is the new threshold.
```

```
IDAStarIter(goalName, world, cost, threshold):
  evaluate goalName(world).
  if goal satisfied:
    return (true, world).
  otherwise:
    possibleActionRuns := generate all valid actions to run
        for world with all possible combinations of arg values
        based on any goal and action optimizations.
    generate children of world by running possibleActionRuns.
    if no children:
        return (false, false).
    currWorld := current worldSnapshot of the obj.
    for each childWorld:
        score := goalName_heuristic(childWorld) + cost + 1.
        if score <= threshold:
          commitToWorld(childWorld).
          (1st, 2nd) :=
            IDAStarIter(goalName, childWorld, cost+1, threshold).
          commitToWorld(currWorld).
          if 1st = true:
            2nd is the solution world, return (true, 2nd).
          otherwise:
            2nd is the cost of child.
            if newThreshold > 2nd: newThreshold := 2nd.
    return (false, newThreshold).
```

Fig. 11. Planning algorithm implementing IDA* search with dynamic optimizations and worlds reasoning

4 Experience

We have developed many small and medium-sized programs in JOHN language that are all available on the project Wiki page. We present a couple of examples here.

4.1 Chess

Our full chess program (partly shown in Figure 10) is only around 300 lines of code, including its ASCII-art graphics. The program looks at all possible moves 2-3 moves ahead and then uses heuristics like those given in Figure 10 to pick a good move. This is a general rule-based implementation of the game with no optimizations. As a result, it is slow and not very smart. On the other hand, it is concise, readable, and extensible by a novice programmer. Any number of heuristics may be added that can make the it smarter, just as a novice player learns more and more strategies as she becomes a stronger player.

4.2 Register Allocation

Pereira et al. introduced a new abstraction for the register allocation problem, by mapping variables and registers to puzzle pieces and boards, then solving the puzzle by linear pattern matching on puzzle pieces [10]. A branch of the industry-level JIT compiler LLVM was implemented with this methodology.

We wrote Pereira's model for the allocation problem in the X86 architecture in no more than 60 lines of code in JOHN. A portion of this code is shown in Figure 12. We then extended the LLVM compiler to optionally plug into our allocator to solve the allocation problem , as seen in Figure 13. Depending on the optimizations selected, the program can find an optimal or fast solution. Although compiling times can be much slower than the original compiler, the running times of some of the benchmarks were reduced when we used the program to find optimal allocation solutions. More importantly, the register allocator program is much more compact, understandable, and easy to extend compared to the original C++ code. A full report for this work can be found in [13].

5 Related Work

Although JOHN can be viewed as a planning tool in the spirit of STRIPS [3], a pioneer automatic planning tool developed in 1971, we designed it as a simplified object-oriented language with integrated planning features. While in terms of performance and capabilities many superior modern tools exist in the planning community [5, 12], JOHN language is unique in that imperative everyday programs can also be constructed using its optimization features. JOHN's feature to add or remove declarative statements is adapted from Prolog [14] family of languages. However, its interactions between the planner and declarative optimizations and heuristics to dynamically guide the search is unique. JOHN and

```
goal Allocator allocation for all v in its variables | v assigned.
optimization Allocator allocation use assign if its roomAvailable.
optimization Allocator allocation use spill if its spillRequired.
optimization Allocator allocation use split.
heuristic Allocator allocation minimize its numSpills.
action Allocator assign Register r Variable v ...
action Allocator spill Variable v ...
action Allocator split Register from Register to ...
dynamic optimization Allocator assign v = its unassignedVars first and
  r = its nextAvailableRegOfLength: v length.
```

Fig. 12. Portion of Register Allocator program

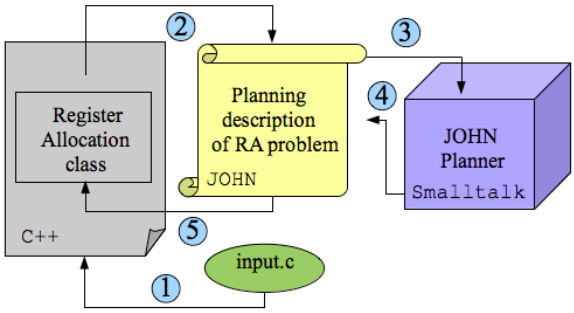

Fig. 13. Extending LLVM Compiler to use a JOHN program to solve the register allocation problem.

PaP are closely related to the idea of a *mixed interpreter* [8, 11] and the Kaleidoscope language [4], which execute programs that have both declarative and imperative parts. The object-oriented semantics and syntax in JOHN are mostly adapted from Smalltalk, while worlds semantics are based on Warth's work [1].

6 Conclusion

We believe that JOHN language is invaluable for education of programming. It aims to avoid the clutter and legacy syntax that predominates the popular languages today, while supporting many of the fundamental ideas in programming such as object orientation and recursion. More features can be easily added. It allows for a natural specifications of heuristics and optimizations, and reasoning power is integrated. These make for a clean, intuitive, and powerful environment to teach programming to students.

Moving real world software in the direction of Programming as Planning is a more long-term goal. We do not expect the methodology to become practical soon enough for many tasks that software developers face today. However, with the emergence of multi-core processors, the possibility of embarking on more declarative and compact mode of computation may eventually become both achievable and necessary.

Acknowledgments

This work was funded by Viewpoints Research Institute. We thank Alan Kay, Todd Millstein, and our colleagues at VPRI for their valuable feedback and support.

References

1. T. K. Alessandro Warth, Yoshiki Ohshima and A. Kay. Worlds: Controlling the scope of side effects. In *Technical Report, Viewpoints Research Institute*, 2010.
2. M. Cavalcanti. Solving air-traffic problems with "possible worlds". In *Proceedings of the Workshop on Executable Modal and Temporal Logics*, pages 144–156, London, UK, 1995. Springer-Verlag.
3. R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Artificial Intelligence 2*, pages 189–205, 1971.
4. B. N. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *ECOOP*, pages 268–286, 1992.
5. H. Kautz and B. Selman. Logic-based artificial intelligence. chapter Encoding domain knowledge for propositional planning, pages 170–209. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
6. R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.
7. J. McCarthy and P. J. Hayes. *Some philosophical problems from the standpoint of artificial intelligence*, pages 26–45. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

8. C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.
9. I. Piumarta and A. Warth. Open, reusable object models. In *S3*, 2008.
10. F. M. Quint ao Pereira and J. Palsberg. Register allocation by puzzle solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM.
11. D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006, New York, NY, USA, 2009. ACM.
12. S. Richter and M. Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. pages 127–177. Journal of Artificial Intelligene Research, Norwell, MA, USA, 2010.
13. H. Samimi. Register allocation via puzzle solving via planning. In *Technical Report, Viewpoints Research Institute*, 2009.
14. O. Stepánková and P. Stepánek. Prolog: A step towards the future of programming. In *Proceedings of the International Summer School on Advanced Topics in Artificial Intelligence*, pages 50–81, London, UK, 1992. Springer-Verlag.