Viewpoints Research Institute

Register Allocation via Puzzle Solving via Planning

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Research Note RN-2009-001

Register Allocation via Puzzle Solving via Planning

Hesam Samimi
UCLA Computer Science Department
University of California, Los Angeles
Email: hesam@cs.ucla.edu

Abstract—**In a recent paper *Register allocation by puzzle solving* 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 [6]. Pereira implemented a branch of the industry-level JIT compiler LLVM with this methodology. The abstraction greatly improved the intuitiveness of the problem and simplified the code, without sacrificing the performance of LLVM extended linear scan algorithm. Inspired by this abstraction we set to use this domain as a test case for the *Programming as planning* methodology we are currently experimenting with [8], in which common programming tasks are written as a planning problem, and a general planner is consulted to get a solution to the problem within a program. Optimizations are easily attached to methods to overcome the deficiencies attributed to using inference systems and general problem solvers. This report presents our initial results of applying this methodology to the specific domain of Register Allocation. A planner with support for optimizations and heuristics is designed as a library in our target programming language, and an allocator program which describes the X86 register allocation problem in a planning form is implemented that calls on the planner to get an optimal solution of the allocation problem.**

Main areas: Register Allocation, Automated Planning

I. Background

Two reasons may contribute to why inference algorithms such as search and propositional logic are not widely used in the general programming community: 1. Implementing search or theorem proving algorithms may require too much work to handle for every programming task 2. It may not be efficient. For a problem where there is a linear time solution, it would not make sense to use a potentially exponential algorithm. However, inference-empowered programs are attractive in several ways: a) Problems of any complexity are equally solved without a change in the program algorithm b) programs are declarative, self-explaining, and simple c) arbitrary number of optimizations are effortlessly attached or removed, and the meaning and optimization parts of programs remain separate. In another paper, we proposed a *Programming as planning* methodology [8] in which programs may avoid solving their sufficiently complex problems by simply describing them to a general problem solver/planner library and asking for a solution.

To address the first problem, we proposed a general planner, which may be search and/or logic based, as a library for the target programming language [5]. Anywhere within a program, an object may call on the planner as a black box problem solver to satisfy any specific goal it may want to satisfy. The object provides the planner a description of the problem, and pass along a copy of its current state, a *world* [9], for the planner to reason about, without interfering with the object while it is waiting. The planner will then solve the problem, and pass back the calling object a solution world, which the object can commit to.

For the question of efficiency, we require a methodology for defining dynamic optimizations and heuristics to try to avoid search/inference as much as possible. For instance, if there is a linear time solution for a given problem, there will be no search at all. With the right optimizations defined, the planner should lead the object directly to its goal state.

To test the *Programming as planning* methodology, we chose the optimal register allocation problem. This is a common task in the programming languages community, and has been shown that its is in fact NP-complete [1]. We implement a heuristic search based planner which accepts attachment of optimizations and heuristics to goals and actions, and then formulate the register allocation problem for the X86 architecture as a planning problem.

Section II briefly describes the register allocation problem, and section III will introduce the project that this work has been based on, and provide a high level look at the plan for the project. In section IV we present the methodologies which may be used to describe the problem and provide optimizations for the planner that will be consulted. An overview of the architecture of the system will be shown in section VI, followed by some of our initial results obtained from benchmarks in section VII. We then follow with related work and conclusions. There is an appendix section which presents our allocator program written in a planning form.

II. Introduction to Register Allocation

In order to compile a source into machine code, the compiler needs to assign as many possible variables to registers, since it is faster to access them in registers vs. memory. However, the registers are limited, and the compiler's allocator needs to be smart about where it decides to store variables at any given point in a program. Each variable has a live range of instructions where it is going to be needed. Given the live ranges of each variable, the allocator aims to fit as many variables in the register banks as possible. Once there is no room for a particular variable in the registers, a variable will need to be *spilled*, stored in memory, in order to make room for variables required to be in register at a particular instruction point.

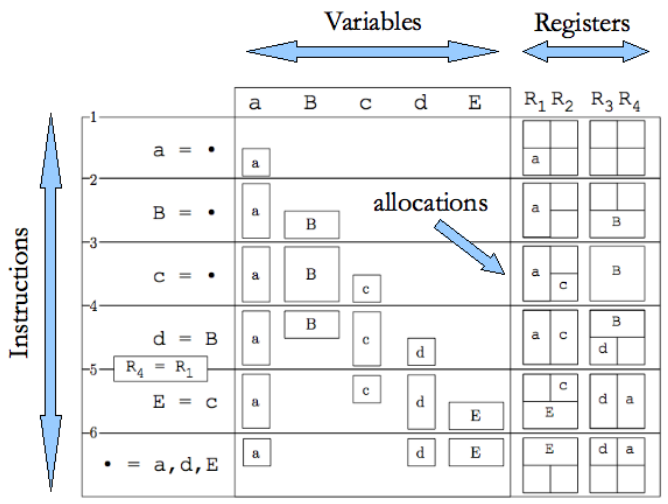

Fig. 1. (left) An example program. (center) Puzzle pieces. (right) Register assignment. (Figure from: *Register Allocation By Puzzle Solving* [6])

In register allocation there are three possible actions to take. In the order of preference, an allocator can either *assign* a register for the variable, *split* the variable's live range to fit in multiple registers, or lastly *spill* in memory, which will hurt the performance of a program. Register allocators aim for minimizing number of splits and and more importantly the number of spills. Figure 1 shows a *basic block* [6] of a program, where variables of different sizes and live ranges are assigned to register banks shown on the right. Variable *a* had to split at the start of the instruction 5 to make room for the larger variable *E*.

III. Plan: Hacking a Compiler

In a recent paper, *Register allocation by puzzle solving*, Pereira et al. introduced a methodology for a compiler to perform register allocation by mapping variables and registers to puzzle pieces and boards, then solving the puzzles by linear pattern matching on puzzle pieces [6]. This new abstraction significantly improved the simplicity and intuitiveness of linear scan algorithm used in LLVM compiler, the Just-in-time (JIT) compiler in the openGL stack of Mac OS 10.5. The puzzle-solving allocator, as LLVM's linear scan, does linear time allocation, since it solves the allocation problem one instruction at a time. This means that it does not guarantee optimal register allocation on general programs. An optimal register allocator however, needs to consider an entire block of instructions together to find an assignment with a minimum number of spills. The left side of Figure 2 shows the difference in the considered scope of a linear (top) allocator vs. one of an optimal allocator (bottom).

In general, it is shown that optimal register allocation is equivalent to graph coloring, and NP-complete [1]. A sub-optimal allocation may *spill* more than necessary number of variables, meaning using memory instead of registers to store them. Pereira et. al's *Puzzle solving register allocator* is a variant of LLVM allocator implemented based on the puzzle solving abstraction. This project is an extension of the puzzle solving allocator, where the allocator is altered to solve the allocation problem by calling a planner library, aiming to find the optimal solution, thus making it a non-linear allocator. Figure 2 demonstrates the background for this project, *Register allocation via puzzle solving via planning*. The main objective is to define enough optimizations to avoid search as much as possible. We aim for comparable compile times, and sometimes better run-times verses the programs compiled with the linear scan allocators. We also will compare the compile time of our planning allocator with other optimal register allocators, namely allocation by graph coloring compilers.

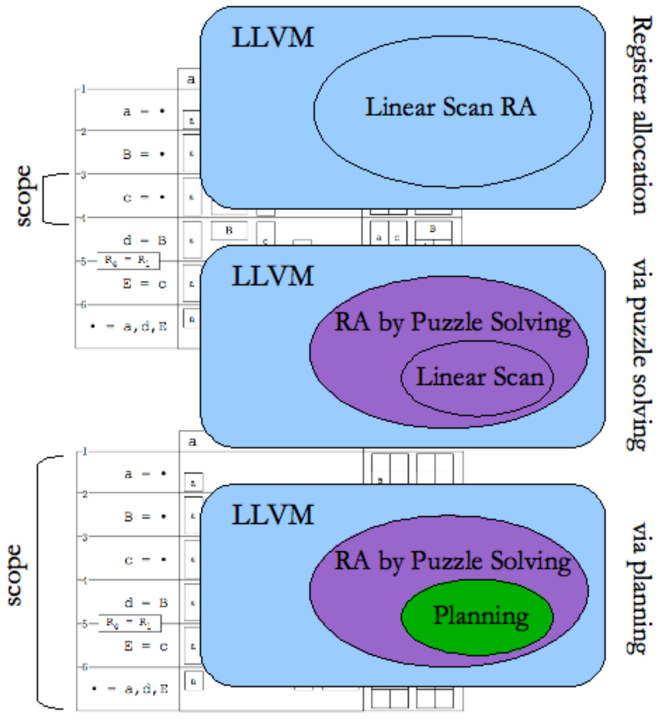

Fig. 2. (left top) a linear scan allocator solves allocation by considering one instruction at a time. (left down) an optimal allocator considers an entire basic block to find a solution. (right top) LLVM. (right middle) LLVM + Puzzle Solving Allocator (right down) LLVM + Puzzle Solving via Planning Allocator.

IV. Methodology

Unlike procedural programs, where usually the entire code is written in the most optimized way, a program describing a planning problem easily separates the real meaning part, from any optimizations that can be turned on or off later on.

A. Describing the RA planning problem

A program using the planner library needs to specify the related objects and their properties, a goal attributed to an object, the choices of actions liable to be useful in reaching that goal, as well as the current state of the object–a *world*. A world captures a snapshot of the object with all its property values at a particular instance [9]. The planner can take this world and apply changes to it without interfering with the current state of the object. Goals are expressed as predicate methods, simply returning a boolean value to specify whether the particular goal is satisfied or not at a given world. The single goal of an allocator object is that all variables have been assigned to registers, as shown in Table 3.

The following sections introduce various parts of program specifying a planning problem in the context of the X86 register allocation. The sample pseudo-code snippets below are selected from the full allocator program, *AllocatorX86*, which appears in the appendix.

1) Classes, Objects: Firstly, we specify the relevant objects in the planning problem. In an object oriented program, this involves naming class names, properties, and messages (methods) they each possess and understand. We then instantiate objects of such classes as needed.

To specify the register allocation problem, we need to define registers, register banks, variables, etc. Table 1 lists the classes of the Allocator program, and Table 2 shows two instances of *Register* class in the X86 architecture, with respective property values for each.

TABLE 1
CLASSES WITH SAMPLE PROPERTIES IN THE ALLOCATORX86 PROGRAM

```
Allocator : (registers variables programSize)
Variable : (name length liveRanges isAssigned)
RegBank : (index allocations)
Register : (length number banks)
```

TABLE 2
SAMPLE OBJECTS IN THE ALLOCATORX86 PROGRAM: A 32-BIT AND A 16-BIT REGISTER

Register Object	length	number	banks
EAX	4	17	[R0, R1, R2, R3]
AX	2	3	[R0, R1]

2) Goals: Table 3 represents the goal method attributed to the *Allocator* object. It is simply a predicate returning whether or not all variables in the program have already been assigned to registers.

TABLE 3
ALLOCATOR GOAL METHOD

```
Allocator allocation
[ ← all Variables assigned? ]
```

3) Actions: Actions are methods that given satisfied preconditions apply a change to the object. There are 3 possible actions an allocator object may need to do in order to reach its allocation goal: *assign*, *split*, and *spill*, presented in Table 4:

In dynamically typed languages, as in *smalltalk* our choice of target language [5], the types of method arguments are unknown until an actual call occurs. The planning involves searching all valid action methods with all possible combinations of argument values. Thus the planner requires the calling

TABLE 4
ALLOCATOR ACTION METHODS, *assign*, *split*, AND *spill*

```
Allocator assign: Reg var: Var
[ if Reg has room for Var:
  [ - mark Reg's banks within live range of Var
     as allocated to Var.
   - remove Var from list of unassigned vars ] ]
```
```
Allocator split: Reg1 to: Reg2 at: InstIdx
[ pick the variable assigned to Reg1 at IndstIdx.
  transfer the variable from IndstIdx on to Reg2 ]
```
```
Allocator spill: Var at: InstIdx
[ free up the register banks which had been
  allocated for Var starting at InstIdx ]
```

object to declare the types of arguments for all action methods. The type of an argument may be stated by specifying a class name, or just an explicit list of candidate values. For example, the *assign:var:* action above, takes two arguments of type *Register* and *Variable* respectively, shown in Table 5.

TABLE 5
ALLOCATOR ACTION METHODS, ASSIGN, SPLIT, AND SPILL

```
Allocator assign: a var: b _argTypes: c
[ ← #(Register Variable) ]
```

B. Attaching Optimizations and Heuristics to Goals

In order to compete with linear scan algorithms, we have to make sure search is used only when there is absolutely no greedy method to make a decision. We require our *Programming as planning* methodology to allow us to easily attach as many optimizations as needed in order to avoid search whenever possible. Possible optimizations to planning problems are generally of two categories: 1) optimization (authors' lack for a better name) and 2) heuristics. Both goal and action methods may have optimizations attached to them. Table 6 summarizes the optimizations scheme for goals and actions.

TABLE 6
DYNAMIC OPTIMIZATIONS

Optimization Type	***Dynamic Optimizations***
Effect	dynamic pruning of search tree
Semantics	which methods (procedures) to explore, with which arguments
Implementation	- a method attached to goal method returning a list of action method names based on predicates - a predicate method attached to action method constraining the argument values
Requires	any search

1) Goal Optimizations: dynamically describe what actions to explore at any given world (node in the search tree). For example, if there is room in the register banks for a variable, there is no need to to explore the spill action. Otherwise, we prefer to split rather than spill any time possible.

TABLE 7
OPTIMIZATION FOR *allocation* GOAL, SPECIFYING WHICH METHOD TO EXPLORE AND WHEN

```
Allocator allocation_optimization
[ if Reg bank has room for next Var:
    [ use #assign:var: ]
  if split helps avoiding a spill:
    [ use #split:to:at: ]
  otherwise:
    [ use #spill:at: ] ]
```

2) Action Optimizations: dynamic constraints for the arguments of the action method. Note that given an optimum solution to the RA problem, where all variables are allocated, the ordering in which variables have been assigned is irrelevant. We then may just fix the ordering in which we assign variables and avoid searching different orderings of variable assignments.

TABLE 8
OPTIMIZATION FOR *assign:var:* ACTION, TO PRUNE IRRELEVANT ARGUMENTS

```
Allocator assign: Reg var: Var _optimization: c
[ ← ( Var = next var to assign ) and
     ( Var length = Reg length ) ]
```

3) Goal Heuristics: Another type of optimization is to guide the planner in deciding which options to explore first, effectively reordering the branches searched. By attaching a heuristic method returning a number representing an estimate of distance to the goal (estimate method may be different at any given state), dynamic heuristic evaluation is used by the planer to prioritize which nodes to explore deeper. This is summarized in Table 9.

TABLE 9
DYNAMIC HEURISTICS

Optimization Type	***Dynamic Heuristics***
Effect	reordering search tree branches
Semantics	which states to explore first
Implementation	a method attached to goal method returning a score for a given state
Requires	Best Fist Search

As an example in the register allocation problem, we may declare a strategy that we are likely to find the optimum solution faster in cases where less number of spilling has been occurred so far. Table 10 shows a heuristic method *allocation_heuristic* attached to the *allocation* goal method, returning the number of spilled variables as a heuristic evaluation function.

TABLE 10
HEURISTIC FOR *allocation* GOAL

```
Allocator allocation_heuristic
[ ← number of spilled variables ]
```

V. PLANNING ALGORITHM

In most problems, the memory space required to store a state of all objects at a given point is very limiting; Equally is the operation of comparing two worlds for duplicity. Therefore, a linear space search best first search algorithm such as *Iterative Deepening A** (IDA*) would be an appropriate search algorithm for our experimental general planner. IDA* works by keeping a threshold cost of a node at any iteration and searching the tree in a *depth first* order, rolling back whenever node of cost exceeding the threshold is generated. Once an iteration is finished without finding a goal node, the threshold is increased to the next minimum seen cost and a fresh new iteration is performed without a need to store previous generations of nodes. This redundancy is usually insignificant in most real problems compared to the gained advantage of not having to store and compare nodes [3]. In the case of register allocation, IDA* is a suitable search algorithm.

An important feature of our IDA* not generally present in the implementations is dynamic rules for node generations. The premise is that for most real world problems, at any given time a different set of actions may be useful in reaching the goal of an object. The proposed planner supports definition of optimizations to dynamically define the liable actions to explore based on the current world. Also for each action, a different set of arguments may be useful at different times. Moreover, the heuristic definitions, strategies for reaching goals quicker, may change from one point to another. The dynamic rules of node generations and branch reordering enables us to optimize the planner for polynomial and pseudo-polynomial problems.

Another essential feature in our search implementation is the use of *worlds* as snapshots of objects at a particular point in time [9]. The planner library cannot interfere with the real state of the object while inferring a solution, since the object can potentially be doing other work while waiting to get back a solution from the planner. We thus require the worlds mechanism, which enables the planner to take a copy of the current world of the object, reason with it without affecting the current world of the object, generating possible worlds stemming from the copy world, and come back with a solution world where the goal of the object is satisfied. The calling object will then be able to commit to this solution world if it wishes to, or may take another action if no solution was found. The planner algorithm is summarized in Table 11.

VI. IMPLEMENTATION

To test their *Register allocation by puzzle solving* methodology, Pereira et al. created a branch of the LLVM compiler source code, written in C++, in which register allocator implements the puzzle solving scheme. This project is based

TABLE 11
PLANNING ALGORITHM IMPLEMENTING IDA* SEARCH, WITH DYNAMIC OPTIMIZATIONS, AND WORLDS REASONING

```
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):
  run 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:
              newThreshol := 2nd.
    return (false,newThreshold)
```

on the puzzle solving allocator branch of LLVM. In our own branch, we have removed the allocation code, and replaced it with a call to our Allocator program, *AllocatorX86* found in appendix, which is a planning description of the problem. This program will simply call our planner, *JOHN* [7], for a solution, then pass the solution assignments back to the LLVM compiler to resume generating the machine code.

Figure 3 is a demonstration of the process. There are two notable advantages in the allocation by planning scheme. Firstly, the planner is used as a black-box, and the means of inference is irrelevant to the calling object. More importantly, there are no system calls. The planner, the RA planning description code, are all internally compiled into C/C++ code. Thus internally, the objects never leave their memory space to get the solution. The inference, in fact, happens within the program itself, without physically appearing in the code, resulting in an equally efficient, yet smaller and simpler program.

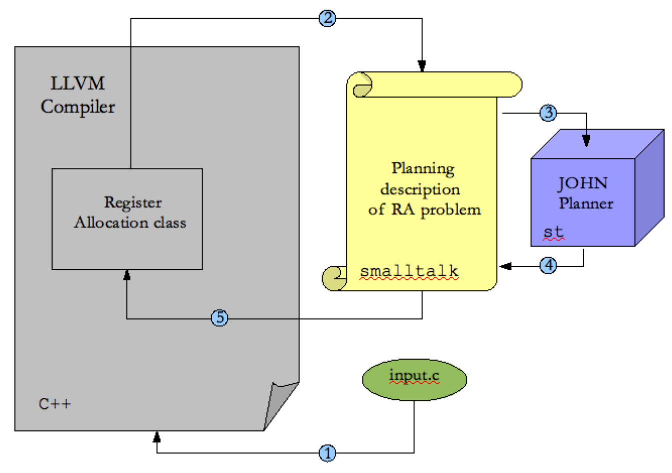

Fig. 3. Register Allocator calls JOHN planner to solve the allocation problem

VII. RESULTS

Our allocator is based on the abstractions of a linear scan algorithm, yet searches for an optimal solution. Thus we need to evaluate the performance of the allocator on two different axes. The compilation time of the algorithm will not be as good as linear ones, yet we compare to them to test the usefulness of our optimizations in place. We also do a straight comparison against an optimal register allocator, which does the allocation by reducing the problem to a graph coloring instance.

Secondly, we evaluate the run times of the programs compiled with our allocator and compare against the puzzle solving allocator. We should see better run times on some cases, when the optimal allocation has incurred less number of spills.

Our original plan was to test our allocator on the LLVM's test suit and regression tests. Unfortunately, given the project deadline, we were unable to resolve a few outstanding bugs to be able to run the full regression tests as of now. As an alternative, we report the results we obtained by running test on a few selected input files.

TABLE 12
SAMPLE RESULTS: PSOLVER = LINEAR PUZZLE ALLOCATOR, PLANNER = PLANNING ALLOCATOR

	Compile Time		**Run Time**	
Test	**PSolver**	**Planner**	**PSolver**	**Planner**
sampla.c	.020s	.120s	.007s	.009s
samplb.c	.177s	.426s	.371s	.175s
samplc.c	.033s	.464s	.176s	.191s
fact.c	.083s	.461s	.157s	.047s
fib.c	.129s	.660s	2.532s	3.091s

VIII. RELATED WORK

A lot of research on register allocation has considered allocation via graph coloring. An *interference graph* of variables is obtained by considering variables as nodes and edges between variables with a overlapping live range [4]. Chaitin et. al showed that there is a corresponding program for any general graph, proving the optimal register allocation for general programs is reducible to graph coloring and NP-complete [1]. It is shown that optimal allocation can be done in time linear in the number of interferences of live ranges, provided the programs are in a *static single assignment* (SSA) form [**?**].

Most of research focuses on detailed optimizations and heuristics, usually polynomial time, techniques, to do each of specific tasks in the register allocation problem: spilling, coloring (assigning), and coalescing (assigning the same register to the two variables involved in a copy assignment, to avoid generation of code for the operation). George and Appel provide a state-of-the-art iterative coalescing technique [2], and several people have introduced extensions to these techniques. Pereira et. al also provided several heuristics and optimizations for spilling and coalescing, some of which was implemented in our allocator as optimizations [6].

IX. Conclusion

It is accepted that declarative programs have the property of being elegant and self-describing. It is also the case that in general they are not as efficient as imperative programs. In this project we set out to apply AI techniques such as heuristic search in a real world programming task, register allocation. A number of promising results were observed, and a few questions remain to be answered as more debugging and testing will be required.

A. Higher quality of code

In the process, many lines of C++ source code was replaced with a small, more understandable, declarative allocator program. By describing problems in a planning form, the semantics and procedures are clearly stated and separated.

B. Flexible optimizations

The goal oriented methodology enables easy attachment and removing of optimizations. We noticed that the greedy techniques and strategies usually implemented within procedural programs can be easily attached to goals and actions in a planning problem. For example, on easy allocation problems, we observed that the optimizations put in place led the planner straight to the the solution, resulting a solution time as efficient as the linear algorithm.

C. Will optimal register allocation be used?

A spill-free allocation translates to faster run-times. Despite this clear advantage, it is not clear whether the trend will side with optimal allocators. As *Just-in-time* (JIT) compilers, which do runtime compiling, are becoming more common, the compile-time performance of programs are considered equally as important. Thus it is likely that polynomial time allocation will be desired, and for an inference based allocator to be useful, it needs to utilize enough optimizations to essentially run in polynomial time. We have observed that the planning allocator can be beneficial, not necessarily to perform optimal allocation, but to simplify the code, and greatly enhance the flexibility of the compiler in utilizing arbitrary number of optimizations.

References

[1] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, USA, 1982. ACM.

[2] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[3] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[4] Jens Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[5] Ian Piumarta and Alessandro Warth. Open, reusable object models. http://www.vpri.org/pdf/tr2006003a_objmod.pdf S3. 2008.

[6] Fernando Magno Quint ao Pereira and Jens 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.

[7] Hesam Samimi. John planner library. http://www.cs.ucla.edu/˜hesam/idst/function/objects/John.st 2009.

[8] Hesam Samimi. Programming as planning. http://www.vpri.org/pdf/m2009001_prog_as.pdf 2009.

[9] Alessandro Warth. Experimenting with programming languages. http://vpri.org/pdf/tr2008003_experimenting.pdf 2008.

Appendix: AllcoatorX86.st - Entire X86 Register Allocator program - written in Pepsi (smalltalk) [5]

TABLE 13
AllocatorX86.st: AllocatorX86

```
{ import: Object }
{ import: John }

"
 This is an example of a Allocator object using the JOHN Planner to solve its
 Register Allocation problem:

 - The Allocator object passes JOHN the methods representing its goal (allocation),
   possible an optimization method for the goal (allocation_optimization),
   actions (assign split), each action has a special method to get types for its
   arguments ([[Register, Variable], [Register, Register]]), and possibly
   optimization methods for each, along with a copy of its current World;
 - JOHN will reason about the world, and return back a solution World (if one exisists),
   which Allocator can commit to if it wishes.

 - X86 Registers:

        EAX:17      ECX:20      EDX:22     ESI:25      EDI:21      EBX:19      EBP:18
    ----------- ----------- ----------- ----------- ----------- ----------- -----------
      AX:3        CX:11       DX:16       SI:84       DI:13       BX:8        BP:6
    -----       -----       -----       -----       -----       -----       -----
  I AL AH       CL CH       DL DH       SIL         DIL         BL BH

variables/registers:
 Variable        class         puzzleType        length(bytes)    category
 32-bit           6                 2               4                1
 16-bit           4                 2               2                1
 8-bit            9                 1               1                1


 float(128-bit?)  2                 4               16               2
 float(128-bit?)  3                 4               16               2
 pseudo-float     11                4               ?                3

banks:        category
general        1
fp1            2
fp2            3

"
Allocator : ObjectPlus (banks banksDic registers variables allocatedRegs unassignedVars
                        spilledVars varNToSplit classToLengthMap classToCategoryMap
                        programSize)
Variable : ObjectPlus (length category number liveRanges usedAtInsts fixedAtInsts
                       assigned assignedRegs)
RegBank : ObjectPlus (section category index allocations)
Register : ObjectPlus (length category number banks)
```

```
Object HS_Allocator_Allocator_newWithProgramSize: arg1
[
    ^Allocator newWithProgramSize: arg1
]

Allocator newWithProgramSize: numInsts
[   | X86Regs |

    " intializes classes, making a collection to keep track of instanciated objects
      of each type: "
    Allocator inits.
    Variable inits.
    RegBank inits.
    Register inits.

    self := Allocator new.
    name := #Al.
    programSize := numInsts.
    allocatedRegs := IdentityDictionary new.
    banksDic := IdentityDictionary new.
    banks := OrderedCollection new.
    registers := IdentityDictionary new.
    variables := IdentityDictionary new.
    unassignedVars := OrderedCollection new.
    spilledVars := IdentityDictionary new.
    classToLengthMap := IdentityDictionary new.
    classToCategoryMap := IdentityDictionary new.
    #((2 16 2) (3 16 2) (4 2 1) (6 4 1) (9 1 1) (11 16 3)) do:
      [:x | classToLengthMap at: (x first) put: (x second).
            classToCategoryMap at: (x first) put: (x third) ].

    Allocator addInstance: self.

    " Register Banks "
    0 to: 31 do: [:n | ( n < 16 or: [ n > 19 ] ) ifTrue:
      [ self newRegBank: ( #R , n ) category: 1
             section: ( n \\ 2 == 0 ifTrue: [ #low ] ifFalse: [ #high ] ) index: n ] ].
    #((2 32 38) (3 39 45)) do:
    [:x | ( x second ) to: ( x third ) do:
      [:n | self newRegBank: ( #R , n ) category: ( x first )
                            section: #high index: n ] ].

    X86Regs :=
         " 32-bit regs in order "
    #((1 ((EAX 4 17 (R3 R2 R1 R0)) (ECX 4 20 (R11 R10 R9 R8))
  (EDX 4 22 (R15 R14 R13 R12)) (ESI 4 25 (R27 R26 R25 R24))
          (EDI 4 21 (R31 R30 R29 R28)) (EBX 4 19 (R7 R6 R5 R4))
          (EBP 4 18 (R23 R22 R21 R20))
         " 16-bit regs in order "
          (AX 2 3 (R1 R0)) (CX 2 11 (R9 R8)) (DX 2 16 (R13 R12)) (SI 2 84 (R25 R24))
          (DI 2 13 (R29 R28)) (BX 2 8 (R5 R4)) (BP 2 6 (R21 R20))
         " 8-bit regs in order "
          (AL 1 2 (R0)) (CL 1 10 (R0)) (DL 1 15 (R0)) (BL 1 5 (R0))
          (AH 1 1 (R0)) (CH 1 9 (R0)) (DH 1 12 (R0)) (BH 1 4 (R0))))
         " Floating point Regs in order "
      (2 ((XMM0 16 96 (R32)) (XMM0 16 97 (R33)) (XMM0 16 104 (R34)) (XMM0 16 105 (R35))
          (XMM0 16 106 (R36)) (XMM0 16 107 (R37)) (XMM0 16 108 (R38))))
         " Pseudo-Floating point Regs in order "
      (3 ((FP0 16 27 (R39)) (FP1 16 28 (R40)) (FP2 16 29 (R41)) (FP3 16 30 (R42))
          (FP4 16 31 (R43)) (FP5 16 32 (R44)) (FP6 16 33 (R45))))).

    X86Regs do:
      [:x | | category regs |
        category := x first.
        regs := x second.
        regs do:
          [:r | self newRegister: ( r first ) length: ( r second )
                     category: category number: ( r third ) banks: ( r fourth ) ] ].

]

" fixme "
Allocator register: regNum allocationAt: idx
[   | alloc |
    ( (registers at: regNum) banks first hasAllocationAtIdx: idx ) ifFalse: [ ^nil ].
    alloc := (registers at: regNum) banks first allocations at: idx.
    ( alloc length = (registers at: regNum) length ) ifTrue: [ ^alloc number ]
                                                     ifFalse: [ ^nil ]
]

" idx should be a beg of instruction index: should be multiple of 2 "
Allocator allocationsAt: idx
[   | res |
    res := IdentityDictionary new.
    allocatedRegs keys do: [:regN | | q1 q2 |
     q1 := self register: regN allocationAt: idx.
     q2 := self register: regN allocationAt: idx + 1.
                             q1 ifTrue: [ res at: q1 put: regN ].
                             q2 ifTrue: [ res at: q2 put: regN ] ].
    ^res
]

Allocator unassignedVars: theVars
[
    unassignedVars := OrderedCollection withAll: theVars
]

Allocator banks [ ^banks ]
Allocator registers [ ^registers ]
Allocator allocatedRegs [ ^allocatedRegs ]
Allocator variables [ ^variables ]
Allocator unassignedVars [ ^unassignedVars ]
Allocator spilledVars [ ^spilledVars ]
Allocator programSize [ ^programSize ]

" qualifications "
Integer symbol
[   | spc |
    spc := self < 10 ifTrue: [ '  ' ]
                    ifFalse: [ self < 100 ifTrue: [ ' ' ] ifFalse: [ '' ] ].
    ^spc , self asString
```

```
]

Integer symbol1
[
    ^( self < 10 ifTrue: [ ' ' , self asString ] ifFalse: [ self asString ] )
]

UndefinedObject symbol [ ^'--' ]
Variable symbol [ ^(self number \\ 1000) asString ]
Symbol symbol [ ^self = #precolored ifTrue: [ 'XX' ] ifFalse: [ self asString ] ]
Integer asAllocationIndex [ ^self / 2 ]
Integer asIntervalIndex [ ^self * 2 ]
Integer asInstIdxStart [ ^self - (self \\ 4) ]

Array atInterval: intv put: val
[
    intv first to: (intv second - 1) do:
      [:i | self at: (i asAllocationIndex) put: val ]
]

RegBank freeAtInterval: intv
[
    intv first to: (intv second - 1) do:
      [:i | (self allocations at: (i asAllocationIndex)) ifTrue: [ ^false ] ].
    ^true
]

RegBank freeUpIntervals: intvs
[
    intvs do: [:intv |
      intv first to: (intv second - 1) do:
        [:i | self allocations at: (i asAllocationIndex) put: nil] ].
    ^true
]

Allocator canAllocate: var
[
    registers keys do:
      [:regN | ( ( registers at: regN ) canAllocate: var ) ifTrue: [ ^true ] ].
    ^false
]

Allocator regToAllocate: var
[
    registers keys do:
      [:regN | ( ( registers at: regN ) canAllocate: var ) ifTrue:
                 [ ^registers at: regN ] ].
    ^false
]

Allocator spillRequired: var
[
    var liveRanges do:
      [:lRange |
        ( self anyBanksFreeAtIdxRangeFrom: (lRange first asAllocationIndex)
       to: ((lRange second - 2) asAllocationIndex) var: var )
```

```
          ifFalse: [ 'spill required for: ' put. var println. ^true ] ].
    ^false
]

Allocator anyBanksFreeAtIdxRangeFrom: s to: e var: var
[
    s to: e do: [:idx | ( self anyBanksFreeAtIdx: idx var: var ) ifFalse:
      [ | instIdx |
        instIdx := idx asIntervalIndex asInstIdxStart.
        varNToSplit := self findVarToSpillAt: instIdx
                    checkLiveIntervals: ( unassignedVars first liveRanges ).
        spilledVars at: varNToSplit put: instIdx.
'spilling ' put. varNToSplit println.
^false ] ].
    ^true
]

Allocator anyBanksFreeAtIdx: idx var: var
[
    banks do: [:bank | ( ( bank category = var category ) and:
                         [ bank freeAtIdx: idx ] ) ifTrue: [ ^true ] ].
    ^false
]

Register canAllocate: var
[
    ( category = var category ) ifFalse: [ ^false ].
    ( length = var length ) ifFalse: [ ^false ].
    banks do: [:bank | var liveRanges do:
      [:lRange | ( bank freeAtInterval: lRange ) ifFalse: [ ^false ] ] ].
    ^true
]

Register deallocateIntervals: intvs
[
    banks do: [:bank | bank freeUpIntervals: intvs ].
    ^true
]

Allocator precolor: regNum intervalStart: intvS intervalEnd: intvE
[
    (registers includesKey: regNum) ifFalse: [ ^false ]. " fixme: no FPX yet... "
    (registers at: regNum) banks do:
      [:bank | bank allocations atInterval: (Array with: intvS with: intvE)
                              put: #precolored ].
    ^true
]


" [ GOAL ] "
Allocator allocation [ ^unassignedVars size = 0 ]

" [ GOAL OPTIMIZATION ] "
Allocator allocation_optimization
[
    | var |
    var := unassignedVars first.
    ^( Array with: ( ( self canAllocate: var )
                     ifTrue: [ #assign:var: ]
                    ifFalse: [ ( self spillRequired: var ) ifTrue: [ #spill: ]
                                                           ifFalse: [ #split:to: ] ] ) )
]

" [ ACTIONS ] "
Allocator assign: reg var: var
[
    var isAssigned ifTrue: [ ^false ].
    ( reg canAllocate: var ) ifFalse: [ ^false ].
    var liveRanges do:
      [:intv | reg banks do: [:bank | bank allocations atInterval: intv put: var ] ].
    self unassignedVars remove: var ifAbsent: false.
    allocatedRegs at: (reg number) put: true.
    var assignedRegs at: (reg number) put: reg.
    'assigned: ' put. var println.
    self draw.
    ^true

]

Allocator split: regBankFrom to: regBankTo
[ | intv splitIntv idx |
    regBankFrom = regBankTo ifTrue: [ ^false ].
    ( regBankFrom section = #high and: [ regBankTo section = #low ] )
      ifFalse: [ ^false ].
    unassignedVars first liveRanges size = 1 ifFalse: [ ^false ].
    intv := unassignedVars first liveRanges first.
    ( regBankFrom allocations at: (intv first asAllocationIndex) ) ifFalse: [ ^false ].
    ( regBankFrom allocations at: (intv first asAllocationIndex) ) = #precolored
      ifTrue: [ ^false ].
    splitIntv := Array with: intv first - 2
                       with: ((regBankFrom allocations at:
      (intv first asAllocationIndex)) liveRanges first second).
    ( regBankTo freeAtInterval: splitIntv) ifFalse: [ ^false ].
    splitIntv first to: splitIntv second do:
      [:i | idx := i asAllocationIndex.
            regBankTo allocations at: idx put: (regBankFrom allocations at: idx) ].
    regBankFrom allocations atInterval: splitIntv put: nil.
    ^true
]

Allocator spill: var
[
    " cut live ranges short for var and re-attempt allocation: "
    var cutLiveRangesAndSelfDeallocateForSpillAt: ( spilledVars at: ( var number ) ).
    ^true
]

" [ ACTION ARG TYPES ] "
Allocator assign: a var: b _argTypes: c [ ^Array with: Register with: Variable ]
Allocator split: a to: b _argTypes: c [ ^Array with: RegBank with: RegBank ]
Allocator spill: a _argTypes: b [ ^Array with: Variable ]
```

```
" [ ACTION OPTIMIZATION ] "
Allocator assign: reg var: var _optimization: c
[
    ^[ ( ( var = unassignedVars first and: [ var length = reg length ] ) and:
         [ var category = reg category ] ) and:
         [ reg = ( self regToAllocate: var ) ] ] " <--- 2nd line fixme "
]

Allocator spill: var _optimization: c
[
    ^[ var number = varNToSplit ]
]


" object snapshot world (not needed with Worlds) "
Allocator worldSnapshot
[ | world res |
  world := IdentityDictionary new.
  world at: #Variables put: IdentityDictionary new.
  world at: #RegBanks put: IdentityDictionary new.
  world at: #unassignedVars put: ( unassignedVars copy ).
  world at: #spilledVars put: ( spilledVars copy ).
  variables keys do:
    [:varN | (world at: #Variables) at: varN put: (variables at: varN) assigned ].
  banks do: [:bank | (world at: #RegBanks) at: bank put: bank allocations copy ].
  ^world
]

" object world commit (not needed with Worlds) "
Allocator commitToWorld: world
[
  self unassignedVars: (world at: #unassignedVars).
  self spilledVars: (world at: #spilledVars).
  variables keys do:
    [:varN | (variables at: varN) assigned: ((world at: #Variables) at: varN) ].
  banks do: [:bank | bank allocations: ((world at: #RegBanks) at: bank) ].
  ^true
]

Allocator newVariable: aName class: aClass liveRangeFrom: aRangeStart to: aRangeEnd
[
    | var |
    var := Variable new.
    var name: aName length: (classToLengthMap at: aClass)
category: (classToCategoryMap at: aClass).
    var addLiveRangeFrom: aRangeStart to: aRangeEnd.
    variables at: aName put: var.
    self addVariable: var.
    Variable addInstance: var.
    ^var
]

Allocator addVariable: var
[
    | idx |
    " keep unassignedVars sorted by start live intervals "
    idx := 0.
    [ unassignedVars size > idx and:
      [ var liveRanges first first >
        ( unassignedVars at: idx ) liveRanges first first ] ] whileTrue:
          [ idx := idx + 1 ].
    unassignedVars insert: var atIndex: idx.
]

Allocator var: varN addUsedInstruction: instIdx
[
    ( variables at: varN ) addUsedInstruction: instIdx
]

Allocator var: varN addFixedInstruction: instIdx
[
    ( variables at: varN ) addFixedInstruction: instIdx
]

Variable name: aName length: aLength category: aCategory
[
  number := aName.
  name := #V , aName.
  length := aLength.
  category := aCategory.
  liveRanges := OrderedCollection new.
  usedAtInsts := OrderedCollection new.
  fixedAtInsts := OrderedCollection new.
  assignedRegs := IdentityDictionary new.
  assigned := false
]

Variable addLiveRangeFrom: aRangeStart to: aRangeEnd
[
  ( aRangeEnd \\ 2 = 0 ) ifFalse: [ aRangeEnd := aRangeEnd + 1 ].
  ( liveRanges size > 0 and: [ aRangeStart = liveRanges last second ] )
     ifTrue: [ ( liveRanges at: ( liveRanges size - 1 ) ) at: 1 put: aRangeEnd ]
    ifFalse: [ liveRanges add: (Array with: aRangeStart with: aRangeEnd) ]
]

Variable cutLiveRangesAndSelfDeallocateForSpillAt: instIdx
[    | res i intvsD intvsFD |
    res := OrderedCollection new.
    i := 0.
    liveRanges do: [:lRange | ( lRange second < instIdx )
      ifTrue: [ res add: lRange.
                i := i + 1 ]
     ifFalse: [ res add: ( Array with: lRange first with: instIdx ).
                intvsD := OrderedCollection
                     with: ( Array with: instIdx with: lRange second)
  withAll: ( liveRanges copyFrom: ( i + 1 ) ).
                intvsFD := self cutIntervalsBasedOnFixedInsts: intvsD starting: instIdx.
                assignedRegs keys do:
                  [:regN | ( assignedRegs at: regN ) deallocateIntervals: intvsFD ].
                liveRanges := res.
                ˆtrue ] ].
]
```

```
" fixme? "
Variable cutIntervalsBasedOnFixedInsts: intvs starting: instIdx
[   | res fixIdxs |
    res := OrderedCollection new.
    fixIdxs := self fixedAtInstsFrom: instIdx.
    fixIdxs ifFalse: [ ^intvs ].
    intvs do:
      [:intv | fixIdxs do:
        [:idx | ( intv covers: idx ) ifTrue: [ res add: ( Array with: intv first
                                                        with: idx ).
                                       ^res ] ].
        res add: intv ].
    ^res
]

Variable fixedAtInstsFrom: instIdx
[    | i |
    i := 0.
    fixedAtInsts do: [:idx | idx < instIdx ifTrue: [ i := i + 1 ]
                                          ifFalse: [ ^fixedAtInsts copyFrom: i ] ].
    ^false
]

Variable addUsedInstruction: instIdx
[
    usedAtInsts add: instIdx
]

Variable addFixedInstruction: instIdx
[
    liveRanges do: [:lRange | ( instIdx == ( lRange first + 2 ) or:
                        [ instIdx == ( lRange second - 2 ) ] )
      ifTrue: [ ^false ] ].
    fixedAtInsts add: instIdx.
    ^true
]

Array covers: idx
[
    ^( idx >= self first asInstIdxStart and: [ idx <= self second ] )
]

Variable nextUsedInstructionAfter: instIdx withinIntervals: intvs
[
    intvs do:
      [:intv | usedAtInsts do:
        [:iIdx | ( iIdx >= instIdx and: [ intv covers: iIdx ] ) ifTrue: [ ^iIdx ] ] ].
    ^false
]

Allocator liveVariablesAt: instIdx
[    | res |
     res := OrderedCollection new.
     variables keys do:
       [:varN | | var s e |
 var := variables at: varN.
         s := var liveRanges first first.
         e := var liveRanges last second.
         ( s < instIdx and: [ e > instIdx ] ) ifTrue: [ res add: varN ] ].
     ^res
]

Allocator findVarToSpillAt: instIdx checkLiveIntervals: intvs
[   | pickedVarN bestIdx |
    bestIdx := 0.
    ( self liveVariablesAt: instIdx ) do:
      [:varN | | var idx |
              var := variables at: varN.
              idx := var nextUsedInstructionAfter: instIdx withinIntervals: intvs.
              idx ifFalse: [ ^varN ].
              idx ifTrue: [ idx > bestIdx ifTrue: [ bestIdx := idx.
                                              pickedVarN := varN ] ] ].
    ^pickedVarN
]


Variable length [ ^length ]
Variable number [ ^number ]
Variable category [ ^category ]
Variable isAssigned [ ^assigned ]
Variable assigned: isAssigned [ assigned := isAssigned ]
Variable usedAtInsts [ ^usedAtInsts ]
Variable fixedAtInsts [ ^fixedAtInsts ]
Variable liveRanges [ ^liveRanges ]
Variable assignedRegs [ ^assignedRegs ]

Allocator newRegBank: aName category: aCategory section: aSection index: aIndex
[   | bank |
    bank := RegBank new.
    bank name: aName category: aCategory section: aSection index: aIndex.
    bank allocations: (Array new: programSize * 2).
    banksDic at: aName put: bank.
    banks add: bank.
    RegBank addInstance: bank.
    ^bank
]

Allocator newRegister: aName length: aLength category: aCategory number: aNumber
          banks: regBankNames
[   | reg numBanks regBanks |
    reg := Register new.
    numBanks := regBankNames size.
    regBanks := Array new: numBanks.
    0 to: ( numBanks - 1 ) do:
      [:n | regBanks at: n put: ( banksDic at: ( regBankNames at: n ) ) ].
    reg name: aName length: aLength category: aCategory number: aNumber banks: regBanks.
    registers at: aNumber put: reg.
    Register addInstance: reg.
    ^reg
]
```

```
Register name: aName length: aLength category: aCategory number: aNumber banks: regBanks
[
  name := aName.
  category := aCategory.
  length := aLength.
  banks := regBanks.
  number := aNumber.
]

RegBank name: aName category: aCategory section: aSection index: aIndex
[
  name := aName.
  category := aCategory.
  section := aSection.
  index := aIndex
]

RegBank hasAllocationAtIdx: idx
[   | alloc |
    alloc := allocations at: idx.
    ^( alloc and: [ alloc ~= #precolored ] )
]

RegBank allocations: allcs [ allocations := Array withAll: allcs ]
RegBank freeAtIdx: idx [ ^( allocations at: idx ) not ]
RegBank allocations [ ^allocations ]
RegBank index [ ^index ]
RegBank section [ ^section ]
RegBank category [ ^category ]

Register banks [ ^banks ]
Register length [ ^length ]
Register category [ ^category ]
Register number [ ^number ]

Allocator run
[ | solutionWorld |

  'Before:' putln.
  self draw.
  solutionWorld := John forObject: self
      satisfyGoal: #allocation
              withActions: (Array with: #assign:var: with: #split:to:)
                           debug: true.
  solutionWorld ifTrue: [ self commitToWorld: solutionWorld ].
  'After:' putln.
  self draw.
]
```