Electronic Notes in Theoretical Computer Science 82 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 17 pages

A Formally Verified Register Allocation Framework

Kent D. Lee [1]

Computer Science
Luther College
Decorah, Iowa USA

Abstract

When using formal methods to generate compilers it is desirable for all levels of the compiler to be formally specified. Typically, register allocation has been thought to be equivalent to graph coloring. Since graph coloring is NP-Complete most algorithms for register allocation have been ad-hoc. This paper presents a framework for register allocation that has been formally verified using an inductive theorem prover.

1 Introduction

Generating compilers from formal language descriptions means targeting a specific machine architecture. Because of the nature of most high-level languages, stack-based target architectures are easy architectures for which to generate code. However, most new architectures are RISC-based and therefore require operands to be in registers.

To ease the assignment of operands to registers, register allocation has typically been performed by assuming an unlimited supply of symbolic registers. Once code generation is completed, ad-hoc methods are used to assign these symbolic registers to the actual, physical registers of the machine [2]. This process resembles graph coloring which is known to be NP-Complete [3]. Therefore it has been thought that register allocation is an NP-Complete problem and hence *must* be done by ad-hoc methods.

This paper presents an algorithm for register allocation that is not NP or NP-Complete. In fact, it is $O(|V|^2)$ where V is the set of symbolic registers being allocated.

The paper begins by presenting an example of a small program to motivate the discussion of register allocation. Then, a formal description of that

[1] Email: leekentd@luther.edu

program using Action Semantics is presented. An assembly language implementation of the program is provided as well, showing how symbolic registers are used in the target code. Next, a property of register interference graphs is presented that allows them to be minimally colored in polynomial time and a proof of correctness is provided. A framework for register allocation, based on the proof is presented in the next section. An example of using the framework to generate code is presented, and finally, the work is compared to other work on register allocation.

2 A Motivating Example

```
let fun f(x)=x+5
    val i = ref 0
in
    i:=input();
    output(1 + ((2 + 3) + f(!i)))
end
```

Fig. 1. a Small program

Consider the program written using a small subset of ML in figure 1. This language will be referred to as the Small language. Given an Action Semantics [8] description for Small [5], the program's corresponding action might look like the one given in figure 10.

Typically, formal semantic descriptions of programs are not directly suitable for code generation in low-level languages [6]. While Action Semantics is chosen as the formal method for language description in this paper, the register allocation framework presented here can be used in any compiler.

Regardless of the chosen formal method for language description, transformations on the description are usually required to get the program's representation ready for code generation. For instance, binding elimination is often performed [9] on actions. Statically known information is usually applied to the formal notation to simplify the program description. Some compilers may employ partial evaluation with respect to some known (static) information [1] to simplify the source program.

Whichever formal language is used, the end result is a transformed description of a program that is in some way more suitable for translation to the intended target architecture. In this paper, the action in figure 10 is transformed by removing statically known bindings and by eliminating unneeded transient values. The resulting action, along with some sort (i.e. type) information is presented in figure 11. The action in figure 11 is ready to be given to a code generator. In the next section this action is given to a code generator which must generate instructions involving registers to implement it.

Quite frequently, the transformed description of a program is in a form

that resembles postfix notation (i.e. operands are evaluated first, followed by the operation itself). For instance, the action in figure 11 is in postfix form. Postfix actions are described in [6]. Stack machines are designed for postfix evaluation. Howevever, register machines are more prevalent. The register allocation framework presented here can be used to emulate a stack machine in registers. This is described in more detail in section 6.

3 Code Generation and Register Allocation

In this section the action in figure 11 is given to a code generator to be translated to a target architecture. Since most target architectures today are RISC, and most RISC architectures have very minor differences (at least at the instruction set architecture level) the particular target architecture is inconsequential. For purposes of discussion consider the MIPS target architecture.

The action given in figure 11 would typically be given to the code generator as an abstract syntax tree, so it is possible to think of generating code incrementally for this action.

Consider the task of generating code for this subaction:

```
| | give 2
| and then
| | give 3
then
| give the sum of the given data
```

As stated in the introduction, code generation is simplified if it is possible to assume an unlimited supply of symbolic registers. Symbolic registers will be denoted by R0, R1, R2, etc. To generate code for this action the number 2 could be loaded into register R7, 3 could be loaded into R8, and R7 and R8 could be added together. This results in the MIPS code:

```
li R7,2
li R8,3
add R7,R7,R8
```

Some things to note about this code:

- The symbolic registers R7 and R8 interfere with each other. That means they cannot be assigned the same physical register.
- At the end of this code R8 contains a value that isn't needed again.
- However, R7 contains a value that will be used later.

This information leads to the construction of a register interference graph. A register interference graph contains vertices representing symbolic registers, and edges exist between vertices when their corresponding symbolic registers interfere with each other. The complete MIPS code for this example program is given in figure 12. The register interference graph for the code is given in

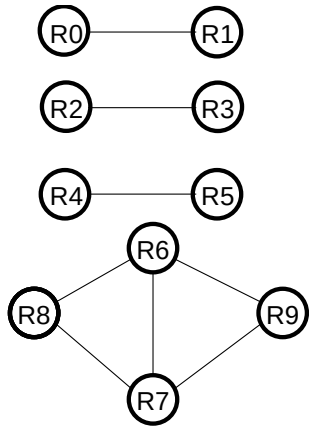

Fig. 2. The register interference graph for the code in figure 12

figure 2.

To complete the code generation, the symbolic registers need to be assigned to physical registers of the machine. From the register interference graph it is fairly easy to conclude that three physical registers (in addition to the physical registers already being used) are required to implement the code in figure 12. In fact, three is the minimum number of registers required given the register interference graph in figure 2. But, how can this be formally verified especially when graphs become more complex?

4 Minimally Coloring Register Interference Graphs

In general, graph coloring is known to be NP-Complete. However, if it is possible to consider only certain types of graphs there may exist polynomial time algorithms to color them. This is the case with register interference graphs, at least the type of register interference graphs considered in this paper. Pittman and Peters say in their book on compiler design that a simple demand policy for allocating symbolic registers seems to work best [10]. This paper helps to formalize that statement. A simple demand based register allocation policy leads to register interference graphs that adhere to what is called the *lifetime property* in graph theory. This property can be formally stated as follows.

Definition: A simple undirected graph $G = (V, E)$, exhibits the **lifetime property** iff $\exists \pi : N \rightarrow V$ so that if $(\pi(i), \pi(k)) \in E$, then $(\pi(i), \pi(j)) \in E$, $i < j < k$ where π is a one-to-one mapping for $N = \{1, 2, ..., |V|\}$.

Informally, this property means that there is an ordering of the vertices such that if two vertices have an edge between them, then all vertices between the first vertex and the last vertex also have edges to the first vertex. In terms of symbolic registers, it means that symbolic registers are created, they carry live values for a while, and then they aren't used again. In other words, register interference graphs that are derived from a simple demand based

register allocation policy.

Note that finding this mapping π, if it exists, for a graph would take exponential time in general, so this does not provide a polynomial time algorithm to solve the graph coloring problem for all graphs for which the lifetime property holds. However, in the case of register interference graphs, the mapping is ready to obtain. It is simply the order in which the symbolic registers were requested. For simplicity, assume from now on that the mapping π is implicitly given as the indices of vertices in V for any graphs with the lifetime property, that is, $\pi(i) = v_i$ for any $v_i \in V$.

Proof that the register interference graphs considered here may be minimally colored in polynomial time is given below. The following proof was first given in a technical report authored by Kent Lee and Hantao Zhang [7]. The proof also leads to the framework for register allocation. The algorithm is implicitly given in the proof of the following theorem, which is done by induction on the number of vertices in V.

Theorem: A simple undirected graph, $G = (V, E)$ exhibiting the lifetime property can be minimally colored in time $O(|V| + |E|)$.

Proof: (by induction on the cardinality of V)

Base case: $|V| = 0$. Clearly, G can be colored by 0 colors in constant time.

Inductive case: Assume, by the induction hypothesis, for all $G = (V, E)$ with $|V| = n$ that G can be minimally colored by k colors in time $O(|E| + |V|)$ if G exhibits the lifetime property. Consider $G' = (V \cup \{v_{n+1}\}, E \cup E')$ with $|V \cup \{v_{n+1}\}| = n+1$ where E' contain only the edges from vertices in V to the vertex v_{n+1}. We must consider three cases for G', which is supposed to exhibit the lifetime property.

Case 1: v_{n+1} has less than k neighbors. Clearly, v_{n+1} can be colored one of the k colors of G that is not one of the colors of the neighbors of v_{n+1}. The neighbor colors of v_{n+1} can be determined by examining the coloring for G and the neighbors of v_{n+1} in E'; this can be done in time $O(|E'|)$. Since no new color is needed, G' is still minimally colored.

Case 2: v_{n+1} has k neighbors. Then, since the lifetime property holds for G', if $(v_i, v_{n+1}) \in E'$ then $(v_i, v_j) \in E$, for $i < j < n + 1$. This means that v_{n+1} forms a $(k + 1)$-clique with its k neighbors in V. But we know from graph theory that a simple undirected graph with a $(k + 1)$-clique cannot be colored in less than $k + 1$ colors. Therefore, we have to assign a new color to v_{n+1}; G' is thus still minimally colored with $k + 1$ colors. This can be done again in time $O(|E'|)$.

Case 3: v_{n+1} has more than k neighbors. Then there exists at least a $(k+1)$-clique in G by the same argument as case 2. But this means that G cannot be colored with k colors. This contradicts our induction hypothesis, so this case is invalid.

Therefore, G' can be minimally colored if G is minimally colored, and we can conclude by induction that all simple undirected graphs exhibiting the

lifetime property may be minimally colored in time $O(|V| + |E|)$ (assuming we know the mapping of the vertices of V). □

The proof given here is the central idea exploited in the framework provided below. In addition, the technical report with Hantao Zhang [7] contains results from proving the same theorem using an automated inductive theorem prover called RRL. The intended property for RRL to prove was: $num\_colors(G) \leq maxclique(G)$. After proving 8 auxiliary lemmas, RRL was able to prove this property in less than 30 seconds on a HP 715 workstation (running the AKCL lisp). Since any graph cannot be colored in less colors than the maximum size clique, it is safe to conclude that the coloring is indeed minimal.

5 A Framework for Register Allocation

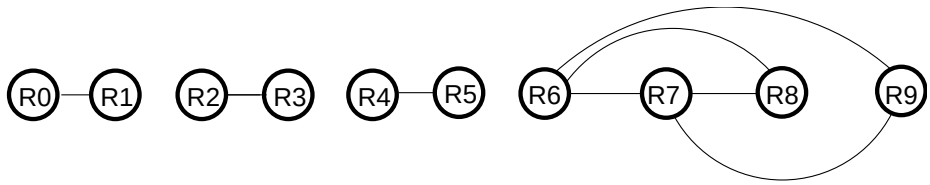

Fig. 3. The register interference graph redrawn

The algorithm for register allocation is based on the simple demand policy described above. With any algorithm or framework, the first step is to arrive at a representation of the problem. In this paper, symbolic register names are R0, R1, R2, etc. Therefore, they may be represented by integers (omitting the R). Register interference graphs are represented by lists of pairs. Each pair consists of two integers. The first integer is the number of the symbolic register. The second number indicates the physical register to which it is assigned (i.e. its color) or negative one (i.e. -1) if the register is still alive.

The list representation is easy to visualize by looking at the graph of figure 2. However, it may help if the graph is redrawn as in figure 3. The list that represents this graph is given below.

$[(9,2),(8,2),(7,1),(6,0),(5,1),(4,0),(3,1),(2,0),(1,1),(0,0)]$

Notice that the second number in each pair is not only the color of the vertex (i.e. its assigned physical register) but it is also the number of neighbors that preceed it in the graph permutation.

Three functions implement this framework. The functions are written in Standard ML. The graph is automatically constructed via the use of these functions.

- The first function, createReg in figure 4, is given a register interference graph and creates a new register by returning a new symobolic register name and a new register interference graph that consists of the given graph with the addition of the new register.

```
val regNum = ref 0;

fun createReg(regList) =
    let val reg = !regNum
    in
        regNum := !regNum+1;
        ("R"^Int.toString(reg),(reg,~1)::regList)
    end
```

Fig. 4. The createReg function

```
exception unknownReg;
exception unFreedReg;

fun unFreedCount([])=0
  | unFreedCount((x,y)::t) =
          if y = ~1
          then 1+unFreedCount(t)
          else unFreedCount(t)

fun freeReg(r, []) =
         (TextIO.output(TextIO.stdOut,
            "Freeing non-existant register "^r^"!\n");
          raise unknownReg)
  | freeReg(r, (regNum,color)::t) =
           if r = "R"^Int.toString(regNum)
           then (regNum,unFreedCount(t))::t
           else if color = ~1
           then
               (TextIO.output(TextIO.stdOut,
                  "Illegally freeing register "^r^
                  " before R"^Int.toString(regNum)^"\n");
                raise unFreedReg)
           else (regNum,color)::freeReg(r,t)
```

Fig. 5. The freeReg function

```
fun offsetRegs([],offset) = []
  | offsetRegs((regNum,color)::t,offset) =
        if color <> ~1
        then (regNum,color+offset)::(offsetRegs(t,offset))
        else (regNum,~1)::(offsetRegs(t,offset))

fun concatRegs(g1, g2) = offsetRegs(g1,unFreedCount g2)@g2
```

Fig. 6. The concatRegs function

- freeReg in figure 5 is called at the end of the lifetime of a symbolic register. It is given a symbolic register and a register interference graph and returns a new register interference graph with the given symbolic register freed.
- concatRegs in figure 6 is called if two register interference graphs are to be merged. This may be needed if code generation has been performed on two parts of a computation (in a bottom-up fashion) and at some later point during code generation the code is merged into one larger computation.

The createReg function uses the -1 (note that in ML -1 is written ~1) to indicate that the graph is not yet complete because registers with color -1 are still alive. The freeReg function uses a result from the theorem. All registers that follow a given register, r, in the register list and are not yet freed must be neighbors of r. Assume there are k such neighbors. Then, if they are neighbors, by the lifetime property they must form a k-clique. But it was proved that this is sufficient to say that with the addition of r they must form a (k+1)-clique. Therefore r should be colored with color k+1. Since colors begin with 0, the unFreedCount returns the next available color.

The freeReg function checks that when a register, r, is freed there are no other registers created after r that are still alive. If there are other live registers that were created after r, freeing r would violate the lifetime property and therefore is not allowed. In effect, this places a first in/first out restriction on register allocation which is another way of thinking of the lifetime property. Section 6 uses the first in/first out property in its implementation.

The concatRegs function can be called to combine register interference graphs at anytime in the future. This is especially useful if your compiler can't guarantee a left to right bottom-up traversal of the abstract syntax tree.

Considering the computational complexity of the functions above the createReg function is $O(1)$. The freeReg function must traverse the register list once for each register that is freed. This is done by calling the unFreedCount function, which is then $O(|V|)$ where V is the set of symbolic registers in the register interference graph. The concatRegs function has similar complexity as it also calls the unFreedCount function. Therefore, the framework performs at least as well as $O(|V|^2)$. Since calls to freeReg occur interspersed with the compilation of the code, in practice there is no noticable wait for register allocation to occur during compilation.

6 An Example Use of the Framework

The framework presented in this paper has been successfully used in practice. In particular, the Action Semantics-based compiler generator called Genesis [5] uses an extension of this framework when targeting the MIPS architecture. The program in figure 1 was compiled by a Genesis generated compiler and the resulting MIPS code (slightly abbreviated due to length) is presented in figure 12. The main goal of Genesis has been to study Action Semantics-based

compiler generation as it applies to sort inference and action transformations. No attempt has been made to make Genesis' compilers produce efficient code.

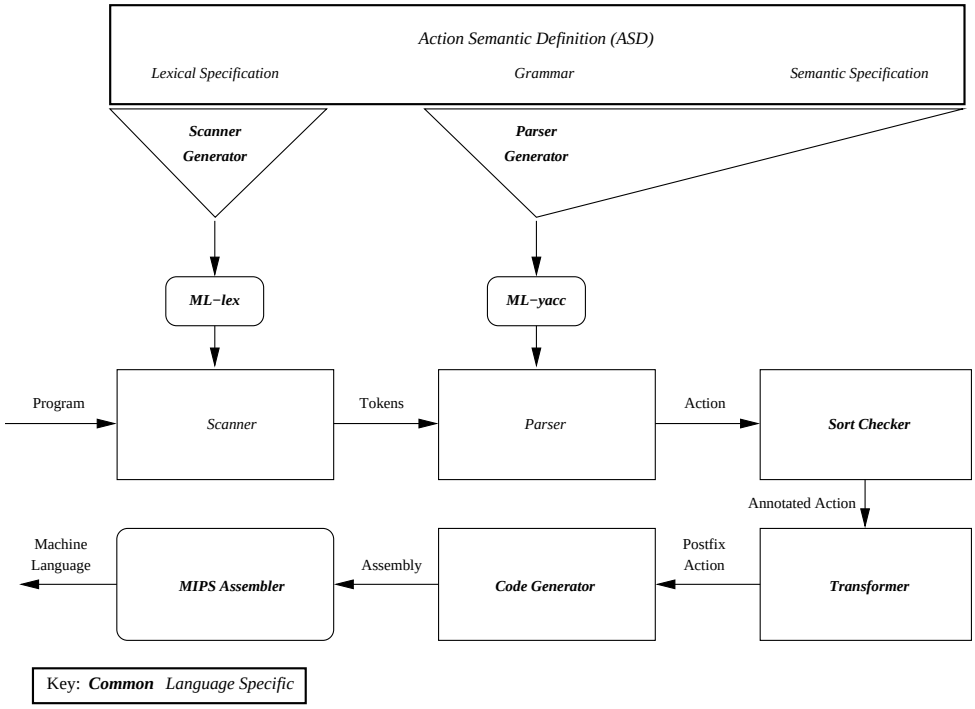

Fig. 7. The Genesis compiler generator

The structure of Genesis is shown in figure 7. Because Genesis is an Action Semantics-based compiler generator, the program in figure 1 is first translated to its action. Then the action is sort checked (i.e. type checking is performed on it) to verify that the action is well-formed. Finally, the action is transformed before it is given as input to the code generator.

Code generation is generally postfix in nature. Operands are evaluated before the operation is applied to the operands. When compiling actions it makes sense to get the action in a form as close to the form of the low-level code as possible. That is the responsibility of the action transformer in Genesis' compilers. It insures that actions are postfix in nature. Postfix actions give their operands and then operate on them. For instance, the action

```
| | give 2
and then
| | give 3
then
| give the sum of the given data
```

begins by evaluating the two operands, by giving 2 and 3, and ends by applying the sum operation to them. A stack machine target architecture is well suited for postfix evaluation. It would be convenient if a register machine could emulate a stack machine, since stack architectures are easy to target when evaluating postfix expressions.

```
val regStack = ref ([]:string list);

exception emptyRegStack;

fun pushReg r =
    (regStack := (r::(!regStack)))

fun popReg() =
    (if (!regStack = []) then raise emptyRegStack else ();
     let val x as (r::rl) = !regStack
     in
         regStack:=rl;
         r
     end)
```

Fig. 8. The pushReg and popReg functions

```
val regList = ref ([]:(int*int) list);

fun getReg() =
    let val result as (r,rl) = createReg(!regList)
    in
        regList:=rl;
        r
    end

fun delReg(r) =
    let val rl = freeReg(r,!regList)
    in
        regList:=rl;
        ()
    end
```

Fig. 9. The getReg and delReg functions

The register framework presented in section 5 can be used to emuluate a stack machine during code generation. With the addition of a few functions, symbolic registers can be "pushed" and "popped" from a register stack using pushReg and popReg. These functions are presented in figure 8. In addition, a couple of functions simplify calls for creating and freeing registers in the framework. The functions getReg and delReg, in figure 9, imperatively create and free registers using the createReg and freeReg functions.

The register framework presented in this paper is used for generating MIPS code in Genesis' compilers. For instance, when the action give 2 is encountered, the code generator responds by calling the getReg function to allocate a register, generating code to put the number 2 in that register, and then

pushing the register onto the register stack using pushReg. The actual code in the code generator, written in ML, looks like this

```
fun gen_action (give'(y, _)) env =
    generate_yielder y env

and generate_yielder (intVal'(i,_)) env =
    let val r = getReg()
    in
      TextIO.output(streamOf env,"\tli "^r^","^Int.toString(i)^"\n");
      pushReg(r)
    end
```

When give the sum of the given data is encountered in the action, the code generator knows that addition is a binary operation. The code generator's code looks like this

```
fun generate_yielder (sum'(y,_)) env =
    let val reg2 = popReg()
        val reg1 = popReg()
    in
      delReg(reg2);
      pushReg(reg1);
      TextIO.output(streamOf env,"\tadd "^reg1^","^reg1^","^reg2^"\n")
    end
```

There should be two operands on the simulated register stack. Upon recognizing the addition operation, the code generator responds by popping two registers from the register stack, generating code to perform the required addition, and then pushing the register containing the result onto the register stack. Notice the register that is no longer needed is freed by calling delReg.

7 Comparison to Other Work

Chaitin's work on register allocation [2] concentrates on allocating *variables* to registers. The focus of that work is studying the liveness range of variables. More recent work by Hendren, et al [4] has extended that work by looking at interval graphs and hierarchical cyclical interval graphs. In each of these approaches nodes in the register interference graph correspond to variables in the program and edges exist between nodes whose corresponding variables interfere with each other. Since variables can exist in virtually any order in a program, there is no structure to the corresponding graphs. The best algorithms for coloring these graphs will always be based on heuristics, assuming that NP-complete is really NP and not P!

However, one possible advantage of these other approaches is that they consider register allocation on a procedure or function level. The approach presented here does not allocate variables to registers. Instead, it focuses on a simpler problem of allocating transient *values* (which may be copies of variables) to registers. The idea of allocating registers for transient values comes from Action Semantics where transients carry values during performance of

an action. By limiting ourselves to values, the resulting register interference graphs have a structure that can be exploited to minimally color them in polynomial time.

Put another way, while other work in register allocation concentrates on allocating registers for an entire procedure or function, the framework presented here concentrates on allocating registers at the statement level, although nothing is preventing it from being used on a procedure or function level.

In comparing the two approaches several things can be said about each approach. The primary advantage of Chaitin's and Hendren's work is in reducing the number of memory accesses that are needed to place values in registers. If a variable is assigned to a register for an entire function or procedure this certainly reduces the number of required loads. Without further work, the approach presented here will certainly result in more load instructions. However, with the advent of large data caches in RISC processors, load instructions don't carry nearly the penalty of earlier CISC architectures.

Write operations may carry a larger penalty in a RISC architecture since writes have the potential to invalidate data in the cache. When considering register allocation on a procedure or function level, spill code will almost inevitably be generated, resulting in writes to memory. However, with the approach presented in this paper, register spilling will happen very infrequently.

8 Conclusion

Register allocation can be expressed in terms of symbolic registers and register interference graphs. If a simple demand policy is employed, register interference graphs adhere to the lifetime property. This paper has shown that such graphs can be minimally colored in polynomial time.

In addition to the somewhat rigorous proof that was given in the paper, the proof was verified by providing it to an automated inductive theorem prover. The theorem prover concluded that graphs that adhere to the lifetime property can be minimally colored. The proof demonstrated the coloring occurs in polynomial time.

The framework presented in this paper takes advantage of the proof. The graphs it produces adhere to the lifetime property, essentially enforcing a first in/first out allocation of registers of the machine.

Future work on this framework includes studying how code generated with it performs when compared to other more conventional register allocation techniques. If it is found that code generated using this framework performs significantly poorer due to an increased number of load instructions, register loads might be suppressed by register coalescing. If registers can be coalesced in such a way that the lifetime property is preserved, some register loads might be eliminated while still being able to minimally color the resulting graphs in polynomial time.

If in experiments register spilling turns out to be an issue, it could also

be studied in the context of preserving the lifetime property. By using this framework as a basis for further research, researchers can either find graph transformations that preserve the lifetime property or understand the trade-offs they are making when abandoning this property in register interference graphs.

Acknowledgement

I would like to thank Hantao Zhang for his help in getting the RRL theorem prover to prove the theorem presented in this paper. Automated inductive theorem proving is still something of an art and Hantao was responsible for discovering the eight axioms needed by RRL to reach its final conclusion.

References

[1] A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FCPA '93)*, Copenhagen, DK, 1993.

[2] Chaitin, Auslander, Chandra, Cocke, Hopkins, and Markstein. Register allocation via coloring. *Computer Languages*, Vol. 6:47–57, 1981.

[3] Garey and Johnson. *A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[4] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *Computational Complexity*, pages 176–191, 1992.

[5] K.D. Lee. *Action Semantics-based Compiler Generation*. PhD thesis, Department of Computer Science, University of Iowa, 1999.

[6] K.D. Lee. Postfix transformations for action notation. In *Proceedings of AS2000*. BRICS Notes Series NS-00-6, 2000.

[7] K.D. Lee and H. Zhang. Formal development of a minimal register allocation algorithm. Technical Report 99-07, University of Iowa, Department of Computer Science, Iowa City, IA, 1999.

[8] P.D. Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26*. Cambridge University Press, 1992.

[9] H. Moura. *Action Notation Transformations*. PhD thesis, Department of Computer Science, University of Glasgow, 1993.

[10] T. Pittman and J. Peters. *The Art of Compiler Design*. Prentice Hall, Englewood Cliffs, NJ 07632, 1992.

```
| | bind “output” to native abstraction of an action
| before
| | bind “input” to native abstraction of an action
hence
| | furthermore
| | | recursively
| | | | bind “f” to closure of the abstraction of
| | | | | furthermore
| | | | | | bind “x” to the given (integer|truth-value)
| | | | | thence
| | | | | | | give (integer|truth-value) bound to “x”
| | | | | | or
| | | | | | | give [(integer|truth-value)]cell bound to “x”
| | | | | and then
| | | | | | give 5
| | | | | then
| | | | | | give the sum of the given data
| | | before
| | | | | allocate [datum]cell
| | | | and then
| | | | | give 0
| | | | then
| | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | and then
| | | | | give the given [datum]cell#1
| | | then
| | | | bind “i” to the given ((integer|truth-value)|[(integer|truth-value)]cell)
| | hence
| | | | | give (integer|truth-value) bound to “i”
| | | | or
| | | | | give [(integer|truth-value)]cell bound to “i”
| | | and then
| | | | | complete
| | | | then
| | | | | enact application of the abstraction of an action bound to “input” to the given data
| | | then
| | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | then
| | | | | give 1
| | | | and then
| | | | | | | | | give 2
| | | | | | | | and then
| | | | | | | | | give 3
| | | | | | | | then
| | | | | | | | | give the sum of the given data
| | | | | | | and then
| | | | | | | | | give (integer|truth-value) bound to “i”
| | | | | | | | or
| | | | | | | | | give [(integer|truth-value)]cell bound to “i”
| | | | | | | | then
| | | | | | | | | give the (integer|truth-value) stored in the given [datum]cell
| | | | | | | | then
| | | | | | | | | enact application of the abstraction of an action bound to “f” to the given data
| | | | | | | then
| | | | | | | | give the sum of the given data
| | | | | then
| | | | | | give the sum of the given data
| | | | then
| | | | | enact application of the abstraction of an action bound to “output” to the given data
```

Fig. 10. A Small program’s action

```
bind "f_101" to the abstraction_(integer)↪(integer)0 of
    give [integer]cell_(0,1)
  and then
    give the given integer
  then
    store the given integer#2 in the given [integer]cell_(0,1)#1
then
    give [integer]cell_(0,1)
  then
    give the integer stored in the given [integer]cell_(0,1)
  and then
    give 5
  then
    give the sum of the given data
and then
    give [integer]cell_(0,0)
  and then
    give 0
  then
    store the given 0#2 in the given [integer]cell_(0,0)#1
and then
    give [integer]cell_(0,0)
  and then
    enact "input" at depth 0_()↪(integer)
  then
    store the given integer#2 in the given [integer]cell_(0,0)#1
then
    give 1
  and then
      give 2
    and then
      give 3
    then
      give the sum of the given data
  and then
      give [integer]cell_(0,0)
    then
      give the integer stored in the given [integer]cell_(0,0)
    then
      enact "f_101" at depth 0_(integer)↪(integer)
  then
    give the sum of the given data
  then
    give the sum of the given data
  then
    enact "output" at depth 0_(integer)↪()
```

Fig. 11. A Small program's transformed action

```
          .data
_prmpt:   .asciiz "? "
          .text
main:     addi $sp,$sp,-40
          sw $ra,4($sp)
          j _f_101
f_101:    addi $sp,$sp,-40
          sw $ra,4($sp)     # store the return address
          sw $t0,0($sp)     # store the access link
          sw $s0,8($sp)     # preserved across function calls
          ...
          sw $a0,36($sp)
          move R0,$sp       # accessing the cell in this way so we can follow access links if necessary
          addi R0,R0,36     # now load the address of the cell
          lw R0,0(R0)
          li R1,5
          add R0,R0,R1
          move $v0,R0
          lw $s0,8($sp)     # restore preserved register
          ...
          addi $sp,$sp,40
          jr $ra            # and return
_f_101:   move R2,$sp       # accessing the cell in this way so we can follow access links if necessary
          addi R2,R2,36     # now load the address of the cell
          li R3,0
          sw R3,0(R2)
          move R4,$sp       # accessing the cell in this way so we can follow access links if necessary
          addi R4,R4,36     # now load the address of the cell
          jal _read
          move R5,$v0
          sw R5,0(R4)
          li R6,1
          li R7,2
          li R8,3
          add R7,R7,R8
          move R9,$sp       # accessing the cell in this way so we can follow access links if necessary
          addi R9,R9,36     # now load the address of the cell
          lw R9,0(R9)
          move $a0,R9       # put the parameter in the parm register
          move $t0,$sp      # set the access link
          jal f_101         # call the function
          move R9,$v0       # save the function result
          add R7,R7,R9
          add R6,R6,R7
          move $a0,R6
          jal _write
          lw $ra,4($sp)     # restore the return address
          addi $sp,$sp,40
          jr $ra
_write:   li $v0,1
          syscall
          jr $ra
_read:    li $v0,4
          la $a0,_prmpt
          syscall
          li $v0,5
          syscall
          jr $ra
          equ R0 $s0    equ R1 $s1    equ R2 $s0    equ R3 $s1    equ R4 $s0
          equ R5 $s1    equ R6 $s0    equ R7 $s1    equ R8 $s2    equ R9 $s2
```

Fig. 12. The MIPS code for the action in figure 11