A Generic Specification for Verifying Peephole Optimizations[*]

A. Dold, F. W. von Henke, H. Pfeifer, H. Rueß

Abt. Künstliche Intelligenz
Fakultät für Informatik
Universität Ulm

Ulmer Informatik-Berichte
Nr. 95–14
December 1995

Abstract

In this paper a generic specification for verifying local optimizations on machine code (peephole optimization) using the specification and verification system PVS is presented. The scheme which provides useful definitions, basic properties and user-defined proof strategies abstracts from the specific instruction set of a machine as well as from its semantics. In addition, we formally represent a stack machine as well as a two-address machine. The general scheme is applied to both machines and local optimizations are formalized and verified using the defined proof strategies.

[*]This research has been funded in part by the German Research Council (DFG) under project "Verifix"

1. Introduction

Peephole optimization is generally understood as the replacement of a sequence of instructions by a semantically equivalent but more efficient sequence. Consider, for example, the addition of constant zero to the content of a register. This is obviously redundant and can be eliminated.[1] Experience shows that optimizers of this kind can tremendously improve the object code [2, 11, 8, 7, 6], especially when the code has been automatically generated by a code generator. Typically, a peephole optimizer works by moving a "window" of the size of two or three consecutive instructions through the object code, and when a pattern is detected it is replaced by the faster sequence. Hence, a peephole optimizer usually works locally and does not incorporate global data-flow knowledge of the machine program. However, some peephole optimizer are extended to consider a restricted form of global optimization [7].

In this paper, we present a generic scheme for formally verifying peephole optimizations and use this scheme to prove correct a set of peephole optimizations for different machines. Although local optimization seems to be an easy task (everybody would accept the "add zero" transformation given above), we demonstrate the importance of a rigorous formal treatment having detected some errors in existing approaches, and making necessary applicability conditions explicit. We show that omitting these conditions would result in incorrect transformations. Our scheme is generic in the sense that we abstract from a specific machine architecture where the optimizations are carried out. It consists of an abstract machine description and a number of definitions based on this description useful for the verification of local transformations such as the concept of *basic blocks* and semantic equality of two basic blocks. Instantiating the scheme, the optimizations can simply be written as triples consisting of two code sequences and an applicability condition. We apply the scheme to two different machine architectures, a stack machine (for intermediate code) consisting of more than 50 instructions and a PDP-11 like two-address machine with several addressing modes. For both machines a number of peephole optimizations are formalized and verified. The purpose of the scheme is to provide a tool for simplifying the verification and administration burden.

We choose the PVS specification and verification system [9] in order to have adequate system support. Its language is based on a higher-order logic with a rich type system including dependent types and semantic subtypes. In addition, it provides tools for analyzing, modifying and documenting theories and proofs together with a powerful sequent calculus based interactive prover. A tutorial [1] provides a detailed comprehensive introduction to PVS. Powerful user-defined proof strategies can greatly improve the level of automation. We have defined some strategies which enable an almost automatic verification of the given transformations. Finally, we hope that this work also serves as an interesting example illustrating the representation of generic specifications in PVS.

The paper is organized as follows: in the following we give an overview of related work. The next section describes the generic peephole optimization scheme. Section 3 and 4

[1]This is only true in cases where no side effects such as a change of the condition codes are considered or if such a modification is irrelevant for the next machine instruction.

present the formalization of a stack machine and a two-address machine, respectively, together with a set of peephole optimizations. Some of the PVS theories are given in the appendix. The complete set of theories and proof scripts are available from the authors.

Related Work

The literature on the construction of peephole optimizers is extensive; however, to the best of our knowledge, no formal treatment of peephole optimization has been published so far.

One of the first machine-independent peephole optimizers has been developed by Davidson and Fraser [2]. Their idea is to simulate pairs of consecutive instructions and replace them, where possible, with an equivalent single instruction. The machine is described by register and memory transfers. Their optimizer is enhanced further by looking for *logically adjacent* instructions instead of *lexically adjacent* ones using a simple data-flow analysis about which resources are accessed or modified by an instruction pair [3], and by the optimizer automatically generating rules from a test set [4].

Tanenbaum's peephole optimizer [11] operates on a stack machine based intermediate code suitable for imperative languages and several machine architectures. Its semantics is given by a pseudo-PASCAL fragment. A large set of optimizations is given in advance in a table including more than 100 rules (pattern/replacement pairs). We show in Sect. 3 that some of the transformations are incorrect or lacking necessary applicability conditions.

Similar work has been carried out by D.A. Lamb [8] constructing a peephole optimizer for VAX11/780 assembly language. Optimizations are described in a pattern language as conditional pattern/replacement rules and an algorithmn using a "peephole window" applies the transformations to the object code.

In [7], the automatic generation of a peephole optimizer from an architectural description of the machine is outlined which additionally performs a restricted form of global data-flow analysis allowing optimizations across basic blocks. The application to the MC68000 is presented. A machine is given in a Lisp based description language. One can specify the addressing modes, the instructions by register transfer statements together with costs for time and space. Based on this architectural description the system automatically generates optimization tables prior to the construction of the compiler. The optimizer then uses the tables to effectively perform the optimizations. The number of consequent instructions to be matched is restricted to two.

Another system [6] automatically generating peephole optimization rules works backwards, i.e. it considers each possible output instruction, decomposes it into parts (for example, a part which modifies a register content and a second part which sets the condition codes), and searches the machine description for all input instructions equivalent to each part. In contrast to [7] this system can identify instruction sequences of arbitrary length that are equivalent to a single instruction. In all these approaches neither a formal machine semantics is provided nor the transformations are formally verified.

In related work on hardware verification, P. Windley uses *generic schemes* to verify microprocessors [12, 13]. A microprocessor is modeled by different levels of abstraction (for example, a macro level which reflects the programmer's view and a micro level specifying the register-transfer level), using a generic state transition system (interpreter) to represent each level. This hierarchical system is then verified by relating the interpreters of subsequent levels. A microprocessor specification of each level consists mainly of four parts:

- a representation of the state,
- a set of state transition functions for each machine instruction, (this corresponds to the *one-step interpreter* described in the next section),
- a selection function which fetches the next instruction to be executed according to the current state,
- and finally, a predicate `I`, the interpreter specification, relating the states before and after the execution of an instruction.

Windley uses the HOL [5] system to represent the generic theory. He has given several instantiations for his theory formalizing and verifying existing microprocessors. In contrast to Windley's work the goal of the work described herein is not to verify existing microprocessors but rather to provide a tool supporting the verification process of local transformations.

2. The Generic Specification

In this section we present a generic scheme for specifying peephole optimizations. Optimizations are specified in a PVS theory that is parameterized by the elements that characterize an abstract machine in a form suitable for verifying optimizations:

1. `inst`: the instruction set of a machine as an uninterpreted type. An instantiation of it normally consists of a (non-recursive) abstract data type where each instruction is given by a constructor.
2. `state`: the machine state, again as an uninterpreted type. Usually, the state is instantiated to a tuple or record type consisting of the register set, the program counter, the memory, status registers, and flags.
3. `admissible`: for each instruction, a predicate that constrains the set of states to which the instruction is applicable. For example, applying a store instruction of a stack machine, which stores the top element of the stack into memory, requires the stack to be nonempty. In PVS we represent `admissible` as a higher-order function from instructions to the domain of predicates over states.
4. `one_step_ip`: For the purpose of local optmizations it suffices to give the semantics of the machine in terms of a one-step interpreter which defines the semantics for each instruction as a state transition function. Hence, we do not provide a

description of the machine's global behavior. [2] The concept of semantic subtypes is used to formalize the condition that the one-step interpreter is only defined for states which are admissible for the specific instruction. This elegantly avoids the use of partial functions and explicit error handling.

5. `jump_inst?`: a predicate on instructions that identifies conditional and unconditional jumps. This is needed since the local optimizations considered here are concerned only with linear code sequences, i.e. code sequence without jumps (*basic blocks*).

These parameters are expressed as formal parameters of a PVS theory (`pred[state]` is an abbreviation for the type `[state -> bool]`):

```
pho_scheme
  [inst : TYPE,
   state : TYPE,
   admissible : [inst -> pred[state]],
   one_step_ip : [o:inst, s:{s1:state | admissible(o)(s1)} -> state],
   jump_inst? : pred[inst]] : THEORY

BEGIN
  [... theory body (see below) ...]
END pho_scheme
```

The following definitions based on the abstract machine description above constitute the theory body of `pho_scheme`.

As stated above the concept of a *basic block* is essential when applying local optimizations. In order to ensure their correctness they can generally be carried out only within a basic block. A program is a sequence of instructions,

```
prg_seq : TYPE = list[inst]
```

and a basic block is a sequence of instructions that has only one entry point and whose transfer mechanism between statements is that of proceeding to the next statement. Here, in order to compare the semantics of two basic blocks it suffices to model a basic block as a sequence of instructions where at most the last instruction is a jump, i.e. a return from a subroutine, or a jump to the next basic block.

```
 basic_block?(l:prg_seq) : RECURSIVE bool =
  CASES l OF
   null : true,
   cons(p1,l1) : IF null?(l1) THEN true
                 ELSE not(jump_inst?(p1)) & basic_block?(l1) ENDIF
  ENDCASES
   MEASURE (LAMBDA (l:prg_seq): length(l))
```

[2] This behavior can be defined as a repeated call of the one-step interpreter depending on the current position of the program counter. One can easily abstract from the behavior of a specific machine. It is a repeated run through the following phases: fetch phase, decoding phase, and execution phase, see also [12].

In PVS only total functions are allowed. For recursive functions one has to provide a well-founded measure for which one has to show that it decreases for each recursive call. Here, one simply uses the length of the instruction sequence.

An interpreter for a basic block can be defined in terms of the given one-step interpreter as a repeated execution of the one-step interpreter starting in an admissible state. A state `s` is *admissible* for a basic block if it is admissible for the first instruction of that block.

```
admis_init(b:(basic_block?), s:state) : bool =
  cons?(b) IMPLIES admissible(car(b))(s)
```

We use an axiomatic specification of the basic block interpreter since we only want to specify the behaviour of the interpreter in admissible states.

```
bb_ip : [bb : (basic_block?), s : state ->  state]

bb_ip_cons : AXIOM
              cons?(b) & admis_init(b, s)
               IMPLIES bb_ip(b, s) = bb_ip(cdr(b), one_step_ip(car(b), s))

bb_ip_null : AXIOM bb_ip(null:(basic_block?), s) = s
```

Semantic equality of two basic blocks can then be defined by means of `bb_ip`. Two basic blocks are (semantically) equal (`eq_bb`) with respect to an applicability condition if the interpreter when started in a state which is admissible for both blocks and where the applicability condition holds results in the same state.

```
eq_bb(b1,b2:(basic_block?), start_cond:pred[state]) : bool =
 (FORALL (s:{s1:state | admis_init(b1,s1) & admis_init(b2,s1) & start_cond(s1)}):
   bb_ip(b1,s) = bb_ip(b2,s))
```

Optimizations on basic blocks can be considered as conditional rewrite rules. We represent them as triples of type `opt_pat` consisting of a given basic block (the *pattern*), the optimized basic block (the *replacement*), and the applicability condition.

```
opt_pat : TYPE = [(basic_block?), (basic_block?), pred[state]]
```

A transformation is correct (`correct_opt_pat?`) if the pattern and the replacement are equal according to the given precondition. A null sequence (`null_seq?`) is a redundant basic block with respect to a given precondition, i.e. this block can be replaced by the empty code sequence, and type `proved_pattern` comprises all correct transformations.

```
correct_opt_pat?(op:opt_pat): bool = eq_bb(proj_1(op), proj_2(op), proj_3(op))

null_seq?(pat:{bb:(basic_block?) | cons?(bb)}, start_cond : pred[state]):bool =
            correct_opt_pat?(pat, null, start_cond)

proved_pattern : TYPE = (correct_opt_pat?)
```

In the following we show that applying a transformation within a basic block under some applicability condition results in a semantically equivalent basic block. In order to prove this we need the notion of an *executable* basic block. A basic block is executable if, when interpreting the block starting in a state where some condition holds, each intermediate state is admissible for the instruction to be applied. The predicate `executable?` is based on an auxiliary predicate `ex_aux?` which is defined recursively on the structure of basic blocks:

```
ex_aux?(b:(basic_block?), s:state) : RECURSIVE bool =
  CASES b OF
   null : true,
   cons(ins, rest) : admissible(ins)(s) & ex_aux?(rest, one_step_ip(ins, s))
  ENDCASES
   MEASURE length(b)

executable?(b:(basic_block?), app_cond:pred[state]) : bool =
 FORALL (s : {s1:state | app_cond(s1)}): ex_aux?(b, s)
```

An optimization can be applied within a basic block `bb` w.r.t. some condition if there exists a "match" of the pattern within `bb` and this match is valid:

```
apl?(op:proved_pattern, bb:(basic_block?), ac:pred[state]) : bool =
  EXISTS (fp, lp : (basic_block?)):
   bb = append(fp, append(proj_1(op), lp)) & applicable?(op, fp, lp, ac)
```

A match is valid (predicate `applicable?`) if

- both the given and the transformed block are basic blocks,
- `fp` is executable w.r.t to the initial condition `ac`, and
- interpreting `fp` up to the beginning of the pattern results in a state which satisfies the applicability condition of the transformation and in which both the pattern and the replacement are executable:

```
applicable?(op:proved_pattern, fp, lp:(basic_block?), ac:pred[state]):bool =
 LET pat = proj_1(op), rep = proj_2(op), app_cond = proj_3(op)
 IN basic_block?(append(fp, append(pat, lp))) &
    basic_block?(append(fp, append(rep, lp))) &
    executable?(fp, ac) &
    (FORALL (s: {s1:state | ac(s1)}):
       LET res_fp = bb_ip(fp, s)
       IN  app_cond(res_fp)
            & executable?(pat, (LAMBDA (st:state): st = res_fp))
            & executable?(rep, (LAMBDA (st:state): st = res_fp)))
```

It remains to show that applying an applicable transformation within a basic block is correct, i.e. both the given basic block and the transformed one are semantically equal with respect to some initial condition `ac`.

```
applicable_equal : THEOREM
 applicable?(op, fp, lp, ac) IMPLIES
   eq_bb(append(fp, append(proj_1(op), lp)),
         append(fp, append(proj_2(op), lp)), ac)
```

The full theory is provided in appendix A. Typechecking this theory results in a number of type correctness conditions (TCC's) which have to be discharged. To manage the proof of `applicable_equal`, we have introduced some additional lemmas. All TCC's and lemmas have been successfully discharged, the proof scripts are available by the authors.

In addition, we provide some proof strategies for this generic theory. A user defined proof strategy consists of basic proof rules together with commands for repeating a command sequence on different branches, rewriting definitions and theories. The first one, called `bb`, proves a program sequence to be a basic block. It simply expands the recursive definition. The second one `pho` proves the correctness of a transformation by rewriting definitions, making type constraints explicit, and applying PVS decision procedures, see appendix B for more details. Most of the transformations given in the next sections are automatically proved using these strategies, some of them require a small amount of additional user interaction.

3. A Stack Machine (SM)

In this section we formally represent a stack machine for intermediate code adapted from [11]. The machine consists of a stack where all arithmetic instructions are carried out, i.e. the operands are fetched from top of the stack and the result is put back to the stack. The machine does not have general registers. Besides arithmetic instructions it provides instructions for loading operands onto the stack and popping them off into memory using several addressing modes (offset, indirect, parameter, direct, ...). Furthermore, instructions for conditional and unconditional jumps, for shifting operands, and special purpose instructions for incrementing, memory clearing, comparisons and block moves are provided. Our formalization includes 56 of 59 instructions. We only have omitted two instructions for rotating bits and the call and return from a procedure. Since all memory locations have to be even, we model `ramadr`, the type of memory locations, as the type of even natural numbers. Memory locations can hold values of type `value`, and the program counter can have values of type `romadr`. Here, we use integer arithmetic.

```
ramadr : TYPE = {n:nat | even?(n)}
romadr : TYPE = nat
value  : TYPE = int
```

It is convenient, though not necessary, to represent the instruction set as an abstract data type (`sm_inst`) since one automatically obtains a full PVS theory (after typechecking has been carried out) together with useful axioms such as the disjointness of the constructors. In addition, PVS's rewriting mechanism efficiently works on abstract data types. The instruction `lop(n)`, for example, indirectly loads the contents of a memory cell `n` onto the stack, and `bra, beq` denote an unconditional and conditional branch, respectively.

```
 sm_inst : DATATYPE
   BEGIN
    lop(lop_adr:ramadr) : lov_inst?
    beq(beq_adr:romadr) : beq_inst?
    bra(bra_adr:romadr) : bra_inst?
     [...]
   END sm_inst
```

A machine state consists of the stack, the memory, and the program counter. We model the memory as a function from memory locations (`ramadr`) to values and the state as a record with selector fields `mem, stk, pc`.[3] We use the theory of parameterized stacks which is predefined in the PVS prelude library.

```
memory = [ramadr -> value]
sm_state = [# mem : memory, stk : stack[value], pc : romadr #]
```

In addition, for each instruction we have to constrain the states in which it is applicable. For example, the `sti(k)` instruction for storing `(k div 2)` elements from stack into memory requires the stack consisting of at least `(k div 2) + 1` elements (predicate `n_stack?`). In addition, it must be ensured that the top element of the stack denotes a valid memory address, i.e. it has to be an even natural number. The higher-order function `sm_admissible` is given by a case analysis on the instruction type:[4]

```
sm_admissible(p:sm_inst) : pred[sm_state] =
 CASES p OF
  add :    LAMBDA (s:sm_state): two_stack?(stk(s)),
  sti(k) : LAMBDA (s:sm_state):
            nonemptystack?(stk(s)) & n_stack?(pop(stk(s)), div2(k)) &
            valid_mem_adr?(top(stk(s))),
  [...]
 ENDCASES
```

The semantics of SM is given by a one-step interpreter which defines the semantics of each instruction separately. Instructions with similar behavior can be grouped together into instruction classes and their effect can then be defined by means of higher-order functions. For example, all binary machine operations (add, sub, mul, xor, ...) have a similar behavior since they fetch two operands from the stack, apply the binary operation and push the result back onto the stack:[5] [6]

[3]Note that in PVS selection of field `mem` from a state `s` is written as `mem(s)` in contrast to the more usual `s.mem` notation.

[4]Predicate `two_stack?` is true if a stack contains at least two elements.

[5]We omit modelling the incrementation of the program counter since within a basic block all instructions are executed sequentially. Only the final value of the pc is relevant: either the program counter points to the beginning of the next block or to some target label of a jump.

[6]In PVS the `WITH` expression is used to update a record at a specific field.

```
binop_sem(s : {s1 : sm_state | two_stack?(stk(s1))},
          bop : [value, value -> value]) : sm_state =
  LET t1 = top(stk(s)), t2 = top(pop(stk(s)))
  IN  s WITH [(stk) := push(bop(t2,t1), pop(pop(stk(s))))]
```

Similarly, the effect of unary operations can be defined. Conditional branch instructions `beq(k), bge(k), ..., bne(k)` pop two operands from the stack, compare them using the associated relation `rel`, and increment the program counter by `k` if the relation is true:

```
branch_sem(s : {s1 : sm_state | two_stack?(stk(s1))},
           rel : [value, value -> bool], k:romadr) : sm_state =
  LET t1 = top(stk(s)), t2 = top(pop(stk(s))),
      newpc = IF rel(t2,t1) THEN pc(s) + k ELSE pc(s) ENDIF
  IN s WITH [(stk) := pop(pop(stk(s))), (pc) := newpc]
```

Analogously, higher-order functions can be defined for compare instructions, and "branch-zero" instructions. The interpreter is then defined by means of these functions.

```
sm_ip(p:sm_inst, s:{st:sm_state | (sm_admissible(p))(st)}) : sm_state =
  CASES p OF
   add : binop_sem(s, (LAMBDA (v1,v2:value): v1 + v2)),
   beq(k) : branch_sem(s, (LAMBDA (v1,v2:value): v1 = v2), k)
   sti(k) : sti_aux(s WITH [(stk) := pop(stk(s))], top(stk(s)), div2(k)),
   [...]
  ENDCASES
```

The meaning of `sti(k)` is defined using an auxiliary recursive function `sti_aux` which stores `(k div 2)` words starting at base address `top(stk(s))`.

```
 sti_aux(s:sm_state, base:ramadr,
         num:{n1:nat | n_stack?(stk(s), n1)}) : RECURSIVE sm_state =
  IF (num = 0) THEN s
  ELSE
   sti_aux(s WITH [(mem)(base + (2 * num) - 2) := top(stk(s)),
                   (stk) := pop(stk(s))], base, num - 1)
  ENDIF
   MEASURE num
```

3.1 Optimizations on the Stack Machine

In [11] more than 100 transformations are given in a pattern/replacement table. We have adopted nearly all transformations, formalized and proved them correct or falsified them. The optimizations are represented as lemmas within PVS theory `sm_pho`. To utilize the generic specification we have to instantiate it with the specific stack machine values:

- `sm_inst`, the (abstract data type) of instructions,
- `sm_state`, the record, consisting of the memory, the stack and the program counter,

- sm_admissible, the admissible functional,
- sm_ip and
- a predicate which denotes SM's conditional and unconditional jumps.

The head of the theory looks as follows:

```
sm_pho : THEORY
BEGIN
  IMPORTING stack_machine,
            pho_scheme[sm_inst, sm_state, sm_admissible, sm_ip,
                       (LAMBDA (oc:sm_inst): beq_inst?(oc) OR
                                             zeq_inst?(oc)
                                             [...])]
    [...]
END sm_pho
```

Tanenbaum has grouped the optimizations into ten different groups. We pick out some characteristic transformations and illustrate their formalization and verification. Most of them can be proved automatically using the peephole optimization strategy pho. Some of them require some additional minor proof steps. However, the main aspect is that we have discovered some transformations being incorrect, and the fact that preconditions which must be established to make the transformation valid have not been provided.

- Constant Folding[7]

  ```
  add_fold : LEMMA
   correct_opt_pat?((: loc(a), loc(b), add :),
                    (: loc(a + b) :),
                    true)
  ```

  Loading constant a and b onto the stack and applying add can be equivalently replaced by loading constant a + b. Here, the applicability condition is given by the constant true function. [8] Another transformation of this group, for example, replaces a negation followed by an addition by an equivalent subtraction.

  ```
   neg_add : LEMMA
    correct_opt_pat?((: neg, add :),
                     (: sub :)
                     true)
  ```

  We have formalized 14 transformations of this group all of them can be proved automatically using the defined proof strategy pho.

[7]The (: ... :) notation is used to represent lists.

[8]Actually, we use type conversions. Since "true" does not have the expected type [sm_state -> bool] a conversion function is applied.

- Operator Strength Reduction
  This group is concerned with the replacement of arithmetic operations by more efficient ones. For example, the replacement of multiplication by two by an efficient left shift operation:

  ```
  mult_shift2 : LEMMA
    correct_opt_pat?((: loc(2), mul :),
                     (: loc(1), shl :),
                      nonemptystack?)
  ```

  Here, the transformation can only be applied in states in which the stack consists of at least one element. All seven transformations of this group can be proved automatically by `pho`.

- Null sequences
  This group mainly deals with redundant instruction sequences. For instance, adding zero is redundant:

  ```
  add_zero : LEMMA
      null_seq?((: loc(0), add :), nonemptystack?)
  ```

  The proof can be established by `pho` plus an applicability of the stack push-eta-axiom given in the ADT-theory of stacks, and an applicability of the extensionality axiom for functions.

  – We have formalized 13 transformations, 6 can be proved by `pho`, 7 require one or two additional steps.

- Combined Moves
  The combined move group tries to combine consecutive push or pop operations into a single one. In this group we have discovered some errors, important preconditions are missing. The transformation below is only valid if the locations given by `m` and `n` are distinct. Omitting the preconditions would result in memory writing conflicts. For instance, trying to prove the transformation `stv_lov_stv` with strategy `pho` the prover stops in a subgoal which can only be solved if the locations given by `m` and `n` are distinct. We have added this precondition, and then have proved the transformation correct.

  ```
    stv_lov_stv : LEMMA
       correct_opt_pat?((: stv(n), lov(m), stv(n+2) :),
                        (: lov(m), sdv(n) :),
                         n /= m)
  ```

  Transformation `lav_blm4` which converts block move instructions to indirect loads and stores requires the stack to be nonempty and its top element must be distinct to location `n + 2`. This precondition is also missing in [11].

```
lav_blm4 : LEMMA
     correct_opt_pat?((: lav(n), blm(4) :),
                      (: loi(4), sdv(n) :),
                      (LAMBDA (s:sm_state): nonemptystack?(stk(s)) &
                                            top(stk(s)) /= n + 2))
```

  - We have formalized all 19 transformations, have corrected two of them, 15 can be proved by pho, 4 require some additional properties about the auxiliary functions sti_aux, loi_aux.

- Commutative Laws

```
  lov_lov_beq : LEMMA
     correct_opt_pat?((: lov(n), lov(n-2), beq(k) :),
                      (: ldv(n-2), beq(k) :),
                     n >= 2)
```

This transformation is valid for all jump instructions.

  - There are 4 transformations, two comprise jumps, so they require an additional case analysis, two can be proved automatically.

- Indirect Moves
The transformations of this group are concerned with replacing indirect moves by more efficient direct ones. For example,

```
lav_loi : LEMMA
   correct_opt_pat?((: lav(n), loi(2) :),
                    (: lov(n) :),
                     true)
```

  - We have formalized all 21 rules, two of them were incorrect, (number 78 and 79), all others can be proved automatically using pho.

- Comparison
These rules deal with comparisons followed by a conditional jump.

```
 tlt_zeq : LEMMA
   correct_opt_pat?((: tlt, zeq(k) :),
                    (: zge(k) :),
                    true)
```

  - We have formalized 4, all of which can be proved by pho plus an additional case analysis.

- Special Instructions
These rules deal with the replacement of an instruction sequence by a special purpose instruction. For example, addition by one can be replaced by an increment instruction.

```
loc1_add : LEMMA
  correct_opt_pat?((: loc(1), add :),
                   (: inc :),
                   true)
```

  - We have formalized 19 transformations, 14 can be proved by `pho`, the others consist of conditional branches and hence require an additional simple case analysis.

- DUP instruction, Reordering
  The DUP group avoids refetching of an operand that is already on the stack, while the reordering group simply reorders the instructions so that other transformations can be applied more easily. For example,

```
stv_lov : LEMMA
  correct_opt_pat?((: stv(n), lov(n) :),
                   (: dup, stv(n) :),
                   true)
```

  - We have formalized all 8 transformations, 7 of them can be proved simply by `pho`, one requires an additional application of the stack-push-eta axiom.

Summarizing the results we have formalized and proved correct 108 transformations, 83 of them can be automatically proved by strategy `pho`, most of the others only require a small additional interaction like for example, a simple case-analysis if conditional jumps are involved. However, the main aspect is that we have discovered some transformations being incorrect and the fact that preconditions which must be established to make the transformations valid have not been provided.

4. A Two-Address Machine (TAM)

The second machine for which we formalize and verify optimizations is a PDP-11 like two-address machine adapted from [2]. It has eight general purpose registers, and the instruction set consists of compare instructions (`test, cmp`), transfer instructions (`mov`), arithmetic instructions (`add, sub, inc, dec`), shift instructions (`asl, asr`), and conditional and unconditional jumps (`bra, beq`). The machine provides several addressing modes:

- Register: `R(n)`, the operand is given in register `n`.
- Autoincrement: `(R(n))+`, the operand is given in memory where register `n` holds the address. The contents of register `n` is automatically incremented by two.
- Autodecrement: `-(R(n))`, decrements first the contents of register `n`. The operand is given in memory at the (decremented) address.

- Index-Address: `x(R(n))`, the operand is given in memory at an indexed address with base `x` and register `n` holds the offset.
- Simple Adress: `x`, the operand is given in memory at location `x`.
- Immediate: `#x`, denotes the constant `x`.

The addressing modes are conveniently represented as an abstract data type `Mode`. The register file is encoded as a function which assigns values to register numbers. We again use integer arithmetic.

```
   regnumber : TYPE = {n : nat | n <= 7}
   reg_file : TYPE = [regnumber -> value]
   memory : TYPE = [adr -> value]

  Mode : DATATYPE
    BEGIN
     reg(reg_num : regnumber) : reg_word?
     autoinc(ai_num : regnumber) : autoinc_word?
     autodec(ad_num : regnumber) : autodec_word?
     index_adr(ind_adr : adr, ind_num : regnumber) : index_word?
     single_adr(s_adr : adr) : single_adr_word?
     lit_value(lit_v : value) : lit_value_word?
    END Mode
```

All modes can be referenced directly or indirectly. This can be encoded as a hierarchical data type using type `Mode`. The type of available addressing modes is then given by datatype `AMode`:

```
 AMode : DATATYPE
   BEGIN
    direct(w_dir : Mode) : direct?
    indirect(w_ind : Mode) : indirect?
   END AMode
```

The machine state is represented as a record consisting of the memory, the register file, the status register which is set according to the result of the instruction, and the program counter.

```
status_value : TYPE = {i:integer | (i = -1) OR (i = 0) OR (i = 1)}

set_status(i:value): status_value =
 IF (i > 0) THEN 0 ELSIF (i = 0) THEN 1 ELSE -1 ENDIF

tam_state : TYPE =
 [# mem : memory, reg : reg_file, flag : status_value, pc : romadr #]
```

Memory addresses are positive while values can also be negative. We must ensure that all referenced memory addresses are positive. Hence we need a predicate for each mode constraining the admissible references. Again we use a functional `admissible_word` which, when given an addressing mode yields a predicate on states. For example, a correct autoincrement reference requires register `n` to hold a positive integer.

```
admissible_word(w:AMode) : pred[tam_state] =
  CASES w OF
   direct(w1) :
     CASES w1 OF
      [...]
       autoinc(n) : (LAMBDA (s:tam_state): reg(s)(n) >= 0),
      [...]
     ENDCASES,
   indirect(w1) :
      [...]
  ENDCASES
```

The semantics of the addressing modes and references is given by function `AMode_sem` which fetches an operand in an admissible state from the given reference. This is again an example how to avoid explicit error handling using PVS's subtype mechanism.

```
AMode_sem(w:AMode, s:{s1:tam_state | (admissible_word(w))(s1)}) : value
  CASES w OF
    direct(w1) :
         CASES w1 OF
          [...]
           autoinc(n) : mem(s)(reg(s)(n)),
          [...]
         ENDCASES,
    indirect(w1) :
         CASES w1 OF
          [...]
           autoinc(n) : mem(s)(mem(s)(reg(s)(n))),
          [...]
         ENDCASES
    ENDCASES
```

As before, we encode the instruction set as an abstract data type.

```
tam_inst : DATATYPE
   BEGIN
     tst(w_tst : (not_dir_immed_mode?)) : tst_inst?
     mov(w_mov1 : AMode, w_mov2 : (not_dir_immed_mode?)) : mov_inst?
     add(w_add1 : AMode, w_add2 : (not_dir_immed_mode?)) : add_inst?
     [...]
   END tam_inst
```

Since not all addressing modes are allowed with all instructions we have to restrict the possible modes using a predicate

```
not_dir_immed_mode?(w:AMode) : bool =
   CASES w OF
    direct(w1) :
     CASES w1 OF
      lit_value(i) : false
     ELSE true
     ENDCASES,
   indirect(w1) : true
  ENDCASES
```

For example, in a `mov(a,b)` operation where `b` gets the value of `a` it is impossible that `b` is a constant.

Instructions can only be executed in admissible states where each operand has to be an admissible reference.

```
tam_admissible(i:tam_inst) : pred[tam_state] =
 CASES i OF
  tst(op) : (LAMBDA (s:tam_state): admissible_word(op)(s)),

  [...]
 ENDCASES
```

The semantics of binary and unary operations again can be defined by means of higher-order functions, for example

```
binop_sem(op1:AMode, op2:(not_dir_immed_mode?),
          s:{s1:tam_state | admissible_word(op1)(s1) & admissible_word(op2)(s1)},
          bop : [value, value -> value]) : tam_state =
     LET res1 = AMode_sem(op1,s),
         res2 = AMode_sem(op2,s),
         newstatus = set_status(bop(res1, res2)),
         res = inst_sem_aux(op2, s, bop(res1, res2))
     IN  res WITH [(flag) := newstatus]
```

The auxiliary function `inst_sem_aux` loads a register or updates the memory at the referenced location with a given value.

```
inst_sem_aux(w:(not_dir_immed_mode?),
             s:{s1:tam_state | (admissible_word(w))(s1)},
             res:value) : tam_state =
   CASES w OF
    direct(w1) :
     CASES w1 OF
      reg(n) : s WITH [(reg)(n) := res],
      [...]
     ENDCASES,
    indirect(w1) :
     CASES w1 OF
      reg(n) : s WITH [(mem)(reg(s)(n)) := res],
      [...]
      ENDCASES
   ENDCASES
```

Function tam_ip then specifies a one-step interpreter for TAM.

```
tam_ip(i:tam_inst, s:{s1:tam_state | tam_admissible(i)(s1)}) : tam_state =
   CASES i OF
    tst(op) : LET result = AMode_sem(op,s)
              IN s WITH [(flag) := set_status(result)],
    add(op1, op2) : binop_sem(op1, op2, s, (LAMBDA (v1,v2:value): v1 + v2)),

    [...]

   ENDCASES
```

4.1 Optimizations on TAM

There are only a few transformations given in [2]. One, for example, deals with the decrementation of a register using the autodecrement mode. Consider the transformation:

```
SUB #2, Ri ; CLR @Ri   --->   CLR -(Ri)
```

First decrementing the contents of register i and then clearing the memory at this location can more efficiently be done by a single instruction using the auto decrementing mode. To formalize this transformation we first have to instantiate the general scheme with the specific TAM values:

```
inst           <--   tam_inst
state          <--   tam_state
admissible     <--   tam_admissible
one_step_ip    <--   tam_ip
jump_inst?     <--   LAMBDA (oc:tam_inst): bra_inst?(oc) OR beq_inst?(oc)
```

The above transformation can then be represented by

```
sub_clr : LEMMA FORALL (n:regnumber):
 correct_opt_pat?(
  (: sub(direct(lit_value(2)), direct(reg(n))), clr(indirect(reg(n))) :),
  (: clr(direct(autodec(n))) :),
  true)
```

The proof is by applying the pho strategy and expanding the auxiliary function inst_sem_aux. We have added a few more optimizations. Adding zero to the content of a register is redundant, i.e. a null sequence. This is not true in general since each instruction changes the value of the status register. In order to make the transformation valid the status value of the initial state must correspond to the status value of the register.

```
add_zero : LEMMA FORALL (n:regnumber):
    null_seq?((: add(direct(lit_value(0)), direct(reg(n))) :),
               LAMBDA (s:tam_state): flag(s) = set_status(reg(s)(n)))
```

The proof is by pho plus an application of the extensionality axiom. Another optimization step deals with the equality of operands. Comparing two equal operands (independent of the addressing modes involved) followed by a conditional branch can be replaced by a simple unconditional branch if in the initial state the status value is set to one.

```
cmp_bra : LEMMA FORALL (op:AMode), (a:romadr):
     correct_opt_pat?((: cmp(op, op), beq(a) :),
                      (: bra(a) :),
                      (LAMBDA (s:tam_state): flag(s) = 1))
```

Here, the proof is by a call of strategy pho.

5. Concluding Remarks

We have outlined how to represent a general scheme for specifying and verifying local optimizations, how to instantiate it to specific machines and how to encode and prove correct a set of local transformations. By detecting some errors, we have demonstrated the importance of a rigorous formal treatment. The PVS system has turned out to be a sutiable tool; in contrast to HOL, for example, it is possible to express abstract schemes directly by means of parameterized theories for which domain specific strategies can be defined. The general scheme can readily be utilized for optimizations on other machines. For example, in addition to the examples given in this paper, we have adopted parts of the formalization of the Tamarack microprocessor given by Windley [10] and have verified some peephole optimizations for Tamarack.

The work presented here illustrates how generic specification and verification can be dealt with in PVS. It is part of a larger effort on constructing verifiably correct compilers; that work makes essential use of similar generic techniques.

References

[1] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA, March 1995. To presented at WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, Boca raton, Florida.

[2] Jack W. Davidson and Christoper W. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[3] Jack W. Davidson and Christoper W. Fraser. Register Allocation and Exhaustive Peephole Optimization. *Software – Practice & Experience*, 14(9):857–865, September 1984.

[4] Jack W. Davidson and Christoper W. Fraser. Automatic Inference and fast Interpretation of Peephole Optimization Rules. *Software – Practice & Experience*, 17(11):801–812, November 1987.

[5] Michael J. C. Gordon. A Proof Generating System for Higher-Order Logic. In P. Subrahmanyam G. Birtwistle, editor, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988.

[6] P. B. Kessler. Discovering Machine Specific Code Improvements. *Sigplan*, 21(7):249–254, 1986.

[7] R. R. Kessler. Peep - an Architectural Description Driven Peephole Optimizer. *Sigplan*, 19(6):106–110, June 1984.

[8] David Alex Lamb. Construction of a Peephole Optimizer. *Software – Practice and Experience*, 11(6):639–647, June 1981.

[9] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[10] Michael Coe Phillip J. Windley. Microprocessor Verification: A Tutorial. Technical Report LAL-92-10, University of Idaho, Department of Computer Science, Laboratory for Applied Logic, 1992.

[11] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, jan 1982.

[12] Phillip J. Windley. A Theory of Generic Interpreters. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*, pages 122–134. Springer-Verlag, May 1993.

[13] Phillip J. Windley. Specifying Instruction-Set Architectures in HOL: A Primer. In Thomas F. Melham and Juanito Camilleri, editors, *Proceedings of the 7th International Workshop on the Higher-Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 440–455. Springer-Verlag, September 1994.

A. The Peephole Optimization Theory

```
pho_scheme  [inst : TYPE+,
             state : TYPE+,
             admissible : [inst -> pred[state]],
             one_step_ip : [o:inst,
                            s:{s1:state | admissible(o)(s1)} -> state],
             jump_inst? : [inst -> bool]] : THEORY
BEGIN

 % -------- code sequence ---------------------------------------------

    prg_seq : TYPE = list[inst]

    null_prg : prg_seq = null[inst]

 % -------- basic block -----------------------------------------------

 basic_block?(l:prg_seq) : RECURSIVE bool =
  CASES l OF
   null : true,
   cons(p1,l1) : IF null?(l1) THEN true ELSE
                   not(jump_inst?(p1)) & basic_block?(l1) ENDIF
  ENDCASES
   MEASURE length(l)


   admis_init(b:(basic_block?), s:state) : bool =
      cons?(b) IMPLIES admissible(car(b))(s)

  b : VAR (basic_block?)
  s : VAR state

 %------  the interpreter for bb's --------------------------------------

   bb_ip : [bb : (basic_block?), s : state ->  state]

   bb_ip1 : AXIOM cons?(b) & admis_init(b, s) IMPLIES
                   bb_ip(b, s) = bb_ip(cdr(b), one_step_ip(car(b), s))

   bb_ip2 : AXIOM bb_ip(null:(basic_block?), s) = s


 % -----------  semantic equality of two bb's -------------------------

 eq_bb(b1,b2:(basic_block?), start_cond:pred[state]) : bool =
  (FORALL (s:{s1 : state | admis_init(b1,s1) & admis_init(b2,s1)
                           & start_cond(s)}):
     bb_ip(b1,s) = bb_ip(b2,s))
```

```
% ------------ executable basic blocks --------------------------------

 % auxiliary function
ex_aux?(b:(basic_block?), s:state) : RECURSIVE bool =
 CASES b OF
  null : true,
  cons(ins, rest) : admissible(ins)(s) &
                    ex_aux?(rest, one_step_ip(ins, s))
 ENDCASES
  MEASURE length(b)

executable?(b:(basic_block?), app_cond : pred[state]) : bool =
  FORALL (s:{s1 : state | app_cond(s1)}): ex_aux?(b, s)

% ------------ optimization pattern ----------------------------------

 opt_pat : TYPE =
  [lhs : (basic_block?), rhs : (basic_block?), ac : pred[state]]

%-----------  correct optimization pattern ---------------------------

 correct_opt_pat?(op:opt_pat): bool =
   eq_bb(proj_1(op), proj_2(op), proj_3(op))

null_seq?(pat:(basic_block?), start_cond : pred[state]) : bool =
  correct_opt_pat?(pat, null_prg, start_cond)

proved_pattern : TYPE = (correct_opt_pat?)

proved_pattern_lst : TYPE = list[proved_pattern]

% ------------ replace a proved pattern within a basic block -----------

l1, l2 : VAR prg_seq
bb, b1, b2, b3, l, r, fp, lp : VAR (basic_block?)
ac,c : VAR pred[state]
op : VAR proved_pattern

% ---  auxiliary lemmas ----------------------------------------------

sublist_is_bb : LEMMA
 basic_block?(append(l1,l2)) IMPLIES basic_block?(l1) & basic_block?(l2)

 bb_app : LEMMA
           executable?(b1, ac) & bb = append(b1,b2) & ac(s)
                IMPLIES
                   bb_ip(bb, s) = bb_ip(b2, bb_ip(b1, s))
 bb_app3 : LEMMA
           executable?(b1, ac) & bb = append(b1, append(b2, b3)) &
            ac(s) & executable?(b2, (LAMBDA (st:state): st = bb_ip(b1, s)))
              IMPLIES
               bb_ip(append(b1, append(b2, b3)), s) =
               bb_ip(b3, bb_ip(b2, bb_ip(b1, s)))

% -------- main lemma: replace l with r in bb ---------------------------

bb_eq : LEMMA
 executable?(b1, ac) & bb = append(b1, append(l, b2))
  & ac(s) & c(bb_ip(b1, s))
    & executable?(l, (LAMBDA (st:state): st = bb_ip(b1, s)))
     & executable?(r, (LAMBDA (st:state): st = bb_ip(b1, s)))
      & basic_block?(append(b1, append(r, b2)))
       &  eq_bb(l, r, c)
            IMPLIES bb_ip(bb, s) = bb_ip(append(b1, append(r, b2)), s)

% -------------- applicablity of a transformation ------------------------

applicable?(op:proved_pattern, fp, lp:(basic_block?),
            ac:pred[state]) : bool =
 basic_block?(append(fp, append(proj_1(op), lp))) &
  executable?(fp, ac) &
   basic_block?(append(fp, append(proj_2(op), lp))) &
    (FORALL (s:{s1 : state | ac(s1)}):
      LET res_fp = bb_ip(fp, s)
      IN  proj_3(op)(res_fp)
           & executable?(proj_1(op), (LAMBDA (st:state): st = res_fp))
            & executable?(proj_2(op), (LAMBDA (st:state): st = res_fp)))

apl?(op:proved_pattern, bb:(basic_block?), ac:pred[state]) : bool =
  EXISTS (fp, lp : (basic_block?)):
    bb = append(fp, append(proj_1(op), lp))
     & applicable?(op, fp, lp, ac)

% -------- main result: correctness of transformation application ---------

applicable_equal : THEOREM
  applicable?(op, fp, lp, ac)
   IMPLIES
    eq_bb(append(fp, append(proj_1(op), lp)),
          append(fp, append(proj_2(op), lp)), ac)

END pho_scheme
```

B. Proof Strategies

- Strategy `bb`

```
(defstep bb ()
 (then* (skosimp)
   (repeat (expand "basic_block?")))
   "(bb) :
      strategy to prove that a machine program is a basic block."
   "~%Applying bb-strategy")
```

- Strategy `pho`

```
(defstep pho (&optional theories rewrites exclude-theories exclude)
(then*
  (skosimp)
  (expand "null_seq?")
  (expand "correct_opt_pat?")
  (expand "eq_bb")
  (skosimp)
  (install-rewrites :defs T theories rewrites exclude-theories exclude)
  (auto-rewrite "bb_ip1" "bb_ip2")
  (typepred "s!1")
  (flatten)
  (assert)
  (repeat (rewrite "bb_ip2"))
  (flatten)
  (assert)
  (apply-extensionality))
 "(pho &OPTIONAL THEORIES REWRITES EXCLUDE-THEORIES EXCLUDE) :
   Sets up auto-rewrites from definitions in the statement,
   from THEORIES and REWRITES,
   and stops rewriting on EXCLUDE-THEORIES and EXCLUDE.
   Then tries to prove the correctness of an optimizing pattern."
  "~%Applying peephole-optimization strategy")
```