Formal Verification of Transformations for Peephole Optimization*

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

Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
{dold,vhenke,pfeifer,ruess}@informatik.uni-ulm.de

To be presented at FME'97, University of Graz, Austria.
September 1997

Abstract. In this paper we describe a formal verification of transformations for peephole optimization using the PVS system [12]. Our basic approach is to develop a generic scheme to mechanize these kinds of verifications for a large class of machine architectures. This generic scheme is instantiated with a formalization of a non-trivial stack machine [14] and a PDP-11 like two-address machine [2], and we prove the correctness of more than 100 published peephole optimization rules for these machines. In the course of verifying these transformations we found several errors in published peephole transformation steps [14]. From the information of failed proof attempts, however, we were able to discover strengthened preconditions for correcting the erroneous transformations.

Keywords: formal verification, transformations, higher-order logic, reusability of specifications.

1 Introduction

Peephole optimization is generally understood as the replacement of a sequence of instructions by a semantically equivalent but more efficient one. Typically, a peephole optimizer works by moving a "window" consisting of two or three consecutive instructions through the object code, and, whenever a peephole pattern is detected replacing it by the "better" sequence. Hence, a peephole optimizer usually works locally and does not incorporate global data-flow knowledge of the machine program. Experience has shown that optimizers of this kind can tremendously improve the object code [2, 7–10, 14], especially when this object code has been automatically generated by a code generator.

On the other hand it is crucial to ensure that the process of peephole optimization indeed replaces sequences of instructions with semantically equivalent

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

ones; one important step in increasing the trustworthiness of peephole optimizers is to verify the correctness of the underlying transformations. Although verification of local optimizations seems to be a relatively easy task, we demonstrate the importance of a rigorous formal treatment, having detected some errors in published transformations for peephole optimization. Moreover, corrected applicability conditions were discovered through formal proof.

In this paper we present a generic scheme for formally verifying peephole optimizations in PVS [13] and use this scheme to formally prove correct sets of peephole optimizations for different machines. The purpose of the scheme is to provide a tool for simplifying the verification and administration burden. Our scheme is generic in the sense that we abstract from a specific machine architecture. It consists of an abstract machine description, a correctness criterion, and a number of definitions based on this description useful for the verification of local transformations, such as conditional transformations.

We applied the scheme to different machine architectures, including 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 these machines, we tried to verify the sets of peephole optimization rules published by Tanenbaum [14], and by Davidson and Fraser [2], respectively. Nearly all of these transformations could be proven mechanically using a simple proof tactic. More interestingly, we found that 5 transformations for the stack machine were erroneous in the general form stated by Tanenbaum [14]. In each of the cases in which we encountered unprovable transformations, however, unsolved subgoals of failed proof attempts pointed us to strengthened admissibility conditions of optimization rules for which the given transformations indeed become equality-preserving.

The paper is organized as follows: in the following we give an overview of related work. Then a brief description of the PVS system is provided. Section 3 presents the generic peephole optimization scheme, and Section 4 describes an instantiation of this scheme for a non-trivial stack machine together with proofs of peephole optimizations for this machine. Finally, Section 5 contains some concluding remarks.

Related Work

There are, of course, quite a number of researchers who have constructed powerful peephole optimizers, for example [2–4, 8–10, 14].

One of the first machine-independent peephole optimizer 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 using a simple data-flow analysis about which resources are accessed or modified by an instruction pair [3], and by automatically generating rules from a test set [4].

Tanenbaum's peephole optimizer [14] operates on a stack-machine-based intermediate code suitable for imperative languages and several machine architec-

tures. A large set of optimizations is given in advance in a table including more than 100 rules (pattern/replacement pairs).

In all these approaches, however, neither a formal machine semantics is provided nor are the transformations formally verified.

McNerney [11] validates transformations on basic blocks by automatically enumerating a set of test programs p, then evaluating both p and its transformation using abstract interpretation. The transformation is meaning-preserving if both p and its transformation are mapped to the same graph in the abstract domain. He implemented an equivalence prover to check this property and used it to validate the register allocation phase of his compiler. It is not clear how his prover can be reused to verify transformations for other target languages.

Windley [15, 16] also uses *generic schemes* but in the realm of hardware verification. He models microprocessors on different levels of abstraction using a generic state transition system (interpreter) to represent each level. The machine specification described in Section 3 is quite similar to Windley's specification of the assembly language level (macro level).

2 A Brief Description of PVS

This section provides a brief overview of PVS. For more details consult [1, 12].

The PVS system combines an expressive specification language with an interactive proof checker. The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `rational`, `real`, among others, and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype {`x:A | P(x)`} consists of exactly those elements of type `A` satisfying predicate `P`. Predicate subtypes are used, for instance, for explicitly constraining the domains and ranges of operations in a specification and to define partial functions. Predicates in PVS are elements of type `bool`, and `pred[A]` is a notational variant for the function type `[A -> bool]`. Sets are identified with their characteristic predicates, and thus the expressions `pred[A]` and `set[A]` are interchangeable. For a predicate `P` of type `pred[A]`, the notation `(P)` is just an abbreviation for the predicate subtype {`x:A | P(x)`}. In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs) in cases where type conflicts cannot immediately be resolved. A large number of TCCs are discharged by specialized proof strategies, and a PVS expression is not considered to be fully type-checked unless all generated TCCs have been proved. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories.

Proofs in PVS are presented in a sequent calculus. The capabilities of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision di-

agrams. Finally, PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies.

3 The Generic Specification

In this section we describe a generic scheme for formalizing peephole optimizations for abstract state machines; the main idea is to identify the minimal requirements that still permit expressing local transformations and correctness thereof.

More precisely, we identified the following set of parameters suitable for verifying peephole optimizations for a large class of abstract state machines:

1. `instr`: the set of (assembly) instructions of an abstract machine is simply given as an uninterpreted type. An instantiation of this uninterpreted instruction type then normally consists of a (non-recursive) abstract data type where each instruction is given by a constructor.
2. `state`: in order to represent the (operational) semantics one has to define the machine state. Again, a concrete interpretation of this state is not needed for developing the generic scheme. Usually, machine states are given as a tuple or record type consisting of the register set, the memory, status registers, and flags.
3. `admissible?`: for each instruction a precondition constrains the set of states in which the instruction is applicable. For example, in order to apply a store instruction of a stack machine which stores the top element of the stack into memory the stack obviously has to be non-empty. In PVS one represents the `admissible?`-predicate as a higher-order boolean function which takes an instruction and yields a state predicate.
4. `effect`: for the purpose of local optimizations it suffices to give the semantics of the state machine in terms of a one-step interpreter which specifies the effect of each instruction. The concept of predicate subtypes is used to formalize the condition that the one-step interpreter is only defined for states which are admissible for a specific instruction.

Altogether, this parameter list leads to the parameterized PVS theory in 1 for the generic peephole optimization scheme; note also that there are no further semantic constraints on these parameters.

```
pho_scheme                                                              1
  [instr        : TYPE,
   state        : TYPE,
   admissible?  : [instr -> pred[state]],
   effect       : [i:instr ->  [(admissible?(i)) -> state]]] : THEORY

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

Given a specific machine, one has to instantiate these formal parameters with machine-specific types and functions. The following definitions, based on this abstract machine description, constitute the body of theory `pho_scheme`.

The application of local transformations is restricted to linear code sequences. We assume that the code sequences considered here do not contain jumps, i.e. conditional and unconditional jumps or returns from a subroutine. Consequently, we disregard the program counter component of the machine state.

An interpreter for a linear code sequence can easily be defined as a repeated execution of the semantics of the single instructions `effect`. Since `effect` can be applied only in states which are admissible for a specific instruction, the interpreter has to be defined as a partial function. We use relations to model partial functions. In PVS, a relation $R \subseteq A \times B$ can be specified as a function mapping elements of type A to a set of elements of type B. Partial functions are then described by restricting the range to sets with at most one element. The recursive function[2] `interprete` 2 takes an instruction sequence `c`, a state `s` and yields a singleton state set which is the result of consecutively executing the instructions if all the instructions of the sequence can be executed, otherwise the empty set denoting undefinedness is returned.

```
% Code : TYPE = list[instr]                                              2

interprete(c:Code)(s:state) : RECURSIVE set[state] =
  CASES c OF
      null      : singleton(s),
      cons(i,r) : IF admissible?(i)(s)
                   THEN interprete(r)(effect(i)(s))
                  ELSE emptyset
                  ENDIF
  ENDCASES
   MEASURE length(c)
```

Obviously, given a sequence `c1 ++ c2` where `++` denotes concatenation, the interpretation of `c1 ++ c2` can be split: the results of interpreting `c1` and `c2` can be relationally composed 3. Note that the operator `++` is overloaded here; the second occurrence in 3 denotes relational composition.

```
interprete_split : LEMMA                                                 3
 interprete(c1 ++ c2) = interprete(c1) ++ interprete(c2)
```

Using this interpreter, two linear code sequences, say `c1` and `c2`, are said to be (semantically) equal if the interpretations of `c1` and `c2`, starting from a state `s`, lead to the same result.

```
==(c1:Code,c2:Code) : [state -> bool] =                                  4
 LAMBDA (s:state): interprete(c1)(s) = interprete(c2)(s)
```

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

Now we have collected all the ingredients to represent peephole optimization rules on linear code sequences and a corresponding correctness criterion. Since transformations consist of an applicability condition, a pattern sequence, and the replacement sequence, they can be encoded as triples of a type `rule` as in 5.

```
rule : TYPE = [# pattern     : Code,                                    5
                replacement : Code,
                condition   : pred[state] #]
```

A transformation is said to be correct (formalized by `correct?` in 6) if the pattern and the replacement are semantically equal under the given precondition.[3] Furthermore, the type `correct_rule` comprises all correct transformation rules.

```
correct?(r:rule) : bool =                                              6
 condition(r) IMPLIES pattern(r) == replacement(r)

correct_rule: TYPE = (correct?)
```

The following theorem expresses the fact that applying an applicable *correct* transformation within a code sequence results in a semantically equivalent sequence.[4]

```
% r: VAR correct_rule, fp,lp: VAR Code                                 7

applicable?(r, fp) : bool =
 FORALL (start:state): interprete(fp)(start) IMPLIES condition(r)

% --- rule application is correct

applicable_equal : THEOREM
 applicable?(r, fp)
   IMPLIES fp ++ pattern(r) ++ lp == fp ++ replacement(r) ++ lp
```

Here, `fp` and `lp` respectively represent the instructions before and after the sequence of instructions to be replaced, and `applicable?(r, fp)` holds if the interpretation of sequence `fp` yields a state satisfying the applicability condition of the transformation rule `r`. The proof of this theorem is by unfolding definitions and using the splitting lemma 3 above.

Using this theorem, a simple peephole optimizer can be specified and proved correct. A function `apply_rule(r, c)` for applying a correct transformation rule `r` within a sequence `c` can be specified using a predicate subtype: the result of applying `apply_rule` is a code sequence `cc` in which an instance of the pattern of `r` has been replaced by the replacement sequence of `r` if there is a match and

[3] Note that the boolean operator `IMPLIES` is overloaded here:
`IMPLIES(p1,p2:pred[state]):bool = FORALL s: p1(s) IMPLIES p2(s)`

[4] The conversion mechanism of the PVS type-checker is used here to include implicit coercions from type `pred[state]` to `bool`.

this rule is applicable; otherwise function `apply_rule` returns `c`. We have carried out a simple implementation of this specification using list functions, but not included it in this presentation.

```
apply_rule(r:correct_rule, c:Code) :                                    8
  { cc:Code | (EXISTS (fp,lp:Code):
                 c = fp ++ pattern(r) ++ lp &
                  applicable?(r, fp) &
                   cc = fp ++ replacement(r) ++ lp)
                    OR (cc = c) }
```

An obvious consequence of the theorem above 7 is the corollary

```
rule_application_correct : COROLLARY c == apply_rule(r, c)              9
```

Our simple peephole optimizer `pho` takes a list of *correct* rules, and a code sequence `c`, and tries to consecutively apply the rules in `rs` to `c`:

```
pho(rs:list[correct_rule], c:Code) : RECURSIVE Code =                   10
 CASES rs OF
  null         : c,
  cons(r,rest) : pho(rest, apply_rule(r, c))
 ENDCASES
  MEASURE length(rs)
```

An easy proof by structural induction shows that our simple peephole optimizer does not change the semantics of a code sequence `c`.

```
pho_correct : THEOREM c == pho(rs, c)                                   11
```

In the rest of this paper we concentrate on the correctness of local transformation rules. In order to establish this correctness, we have developed a proof strategy 12.

```
(defstep pho (&optional theories rewrites exclude)                      12
 (THEN*
  (GRIND :defs ! theories rewrites exclude)
  (REWRITE "singleton_lem")
  (REPEAT (APPLY-EXTENSIONALITY :HIDE? T))
  (REDUCE))
   "(pho &OPTIONAL THEORIES REWRITES EXCLUDE) :
    Sets up auto-rewrites from definitions in the statement,
    from THEORIES and REWRITES,
    and stops rewriting on EXCLUDE.
    Then tries to prove the correctness of an optimizing pattern."
  "~%Applying peephole-optimization strategy")
```

The strategy can be called with additional parameters for installing and excluding definitions and theorems for automatic rewriting. `THEN*` is a strategy

which applies the first command that follows to the current goal; the rest of the commands are then applied to each subgoal generated by the first command application. `GRIND` is the most powerful built-in strategy. It combines rewriting with propositional simplification using BDD's and decision procedures. Most of the optimization steps presented in the next section can be proved simply with `GRIND`. However, some additional effort is required for a few of them. Unfolding the definition of the interpreter, two singleton state sets have to be compared in the final proof state. This can be reduced to proving the equality of the states using the corollary `singleton_lem` 13.

```
singleton_lem : COROLLARY                                          13
 s1 = s2 IMPLIES singleton(s1) = singleton(s2)
```

Repeatedly applying the extensionality axiom and then applying `REDUCE` may finish the proof. All but a few optimizations given in the next section can be proved automatically using this strategy.

4 Verification of Stack Machine Optimizations

In this section we formally represent a non-trivial stack machine with more than 50 instructions as an instance of the generic scheme developed in Section 3. Then, we describe the proof efforts for proving the correctness of the optimization patterns as listed by Tanenbaum [14] for this architecture.

The machine consists of a stack on which all arithmetic instructions are executed, i.e. the operands are fetched from top of the stack and the result is put back onto 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, etc). 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 all instructions except those dealing with transfer of control (i.e. jumps, procedure calls, returns etc.)

It is convenient, though not necessary, to represent the instruction set as an abstract data type (`sm_inst` in 14). The instruction `blm(n)`, for example, moves a memory block, `lop(n)` indirectly loads the contents of a memory cell `n` onto the stack, and `sti(k)` stores `(k div 2)` elements from the stack into memory where the base memory address is taken from the top of the stack.[5]

[5] Since memory addresses have to be even we model `ramadr` as the type of even natural numbers `{n:nat | even(n)}`. Type `value` denotes integers.

```
sm_inst : DATATYPE                                                    14
 BEGIN
   ...
   blm(blm_n:ramadr)     : blm?
   lop(lop_adr: ramadr)  : lov?
   sti(sti_n:ramadr)     : sti?
   ...
 END sm_inst
```

The machine state consists of the stack and the memory. Since we do not model jumps the program counter is not included in the state. In 15, memory is modeled as a function from memory locations (`ramadr`) to values and the state as a record with selector fields `mem, stk`; the theory of parameterized stacks is defined as abstract datatypes in the obvious way.

```
memory  : TYPE  = [ramadr -> value]                                   15
state   : TYPE+ = [# mem : memory, stk : Stack #]
```

In addition, for each instruction one has to constrain the states in which it is applicable. For example, the `sti(k)` instruction for storing `(k div 2)` elements from the stack into memory requires the stack consisting of at least `(k div 2) + 1` elements (predicate `n_tops?`). Also, 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:[6]

```
sm_admissible(i:sm_inst) : pred[state] =                              16
 LAMBDA (s:state):
  CASES i OF
   ...
   add    : twotops?(stk(s)),
   sti(k) : nonempty?(stk(s)) &
            n_tops?(pop(stk(s)), div2(k)) &
            top(stk(s)) >= 0 &
            even(top(stk(s))),
   ...
 ENDCASES
```

The semantics of the stack machine is given by a one-step interpreter which defines the effects 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: they fetch two operands from the stack, apply the binary operation, and push the result back onto the stack 17. [7]

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

[7] In PVS the `WITH` expression is used to denote updating a record at a specific field.

```
% bop : VAR [value,value -> value]                                    17

binop_sem(bop)(s:{s1:state | twotops?(stk(s1))}) : state =
  LET t1 = top(stk(s)), t2 = top(pop(stk(s))) IN
   s WITH [(stk) := push(bop(t2,t1), pop(pop(stk(s))))]
```

The effect of unary operations can be defined similarly. Compare instructions pop the top operand from the stack, compare it with 0 using the associated relation `rel`, and push 1 or 0 onto the stack if the comparison yields true or false, respectively 18.

```
% rel : VAR [value,value -> bool]                                     18

comp_sem(rel)(s:{s1:state | nonempty?(stk(s1))}) : state =
 LET t = top(stk(s)),
     newstk = (IF rel(t, 0) THEN push(1, pop(stk(s)))
               ELSE push(0, pop(stk(s)))
               ENDIF)
 IN s WITH [(stk) := newstk]
```

The one-step interpreter `sm_ip` 19 is then defined using these functions.

```
sm_ip(i:sm_inst)(s:{s1:state | (sm_admissible(i))(s1)}) : state =     19
  CASES i OF
   ...
   add    : binop_sem(LAMBDA v1,v2: v1 + v2)(s),
   teq    : comp_sem(LAMBDA v1,v2: v1 = v2)(s),
   sti(k) : sti_aux(s WITH [(stk) := pop(stk(s))], top(stk(s)), div2(k)),
   ...
  ENDCASES
```

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

To utilize the generic specification from Section 3 for this stack machine, the following actual parameters are used for the formal parameters stated in 1:

- `sm_inst`, the (abstract data type) of instructions,
- `state`, the record, consisting of the memory, and the stack,
- `sm_admissible`, the admissible functional, and
- `sm_ip`, the effect function.

4.1 Correctness-Preserving Optimizations for the Stack Machine

In [14] more than 100 transformations are given in a pattern/replacement table. We have examined nearly all transformations, formalized and proved them correct or falsified them. We have omitted only transformations containing jump instructions and instructions concerning procedures.

```
tan1   : LEMMA correct?((# pattern     := (: loc(a), loc(b), add :),
                           replacement := (: loc(a + b) :),
                           condition   := true #))

tan17  : LEMMA correct?((# pattern     := (: neg,add :),
                           replacement := (: sub :),
                           condition   := true #))

tan23  : LEMMA correct?((# pattern     := (: loc(2), mul :),
                           replacement := (: loc(1), shl :),
                           condition   := true #))

tan32  : LEMMA correct?((# pattern     := (: loc(0), add :),
                           replacement := null,
                           condition   := nonempty? #))

tan65  : LEMMA correct?((# pattern     := (: lav(n), blm(4) :),
                           replacement := (: loi(4), sdv(n) :),
                           condition   := LAMBDA s: nonempty?(stk(s)) &
                                          top(stk(s)) /= n + 2 #))

tan123 : LEMMA correct?((# pattern     := (: div, neg :),
                           replacement := (: neg, div :),
                           condition   := true #))
```

Fig. 1. Some Peephole Optimizations for Stack Machine

An excerpt of the list of peephole optimizations can be found in Fig. 1. Consider, for example, rule `tan1` in Fig. 1.[8] This rule is always applicable, and permits replacing the *pattern* part with the *replacement* part, since loading constants `a` and `b` onto the stack followed by an application of `add` is equivalent to simply loading the constant `a + b`. The pattern in `tan32` is a redundant code sequence since adding 0 to a value is redundant. However, this rule is only applicable if the stack is non-empty. We found that the simple strategy `pho` described in the preceding section suffices to prove the vast majority of peephole optimizations fully automatically.

In some cases, proof attempts failed and, altogether, we have discovered 5 erroneous transformation rules in Tanenbaum's [14] list of peephole optimizations. Consider, for example, the rule `tan62` in [20].

[8] The conversion mechanism of the PVS type-checker is used here to include implicit coercions from type `bool` to `pred[state]` and from type `pred[Stack]` to `pred[state]`

```
tan62 : LEMMA correct?(                                                    20
     (# pattern     := (: stv(n), lov(m), stv(n + 2) :),
        replacement := (: lov(m), sdv(n) :),
        condition   := LAMBDA s: n /= m #))
```

This transformation tries to combine consecutive push and pop operations into a single one. `stv(n)` stores the top element at location `m`, `lov(n)` pushes the content of location `n` onto the stack, and `stv(n)` stores the two top elements at locations `n + 2` and `n`, respectively. Tanenbaum lists this rule without the precondition `n /= m`. Trying to prove this erroneous transformation with our specialized proof strategy, the prover stops in a subgoal which can only be solved if the locations given by `m` and `n` are distinct, since omitting this precondition results in a memory writing conflict:

```
{-1} n!1 = m!1
{-2} ...
  |-------
{1}  mem(s!1) WITH [(n!1)     := top(stk(s!1))]
              WITH [(2 + n!1) := top(stk(s!1))]
     =
     mem(s!1) WITH [(2 + n!1) := mem(s!1)(m!1),
                    (n!1)     := top(stk(s!1))]
{2} ...
```

A similar inspection leads to strengthened preconditions for some of the other incorrect transformations. In addition, some transformations have been corrected by changing one or more instructions in the pattern or the replacement. Note, however, that the discovery of strengthened preconditions from failed proof attempts is not automated and requires a close analysis of the unsolved subgoals.

Summarizing the results, we have formalized and proved correct 108 transformations (out of 123 in [14]), 101 of them are proved automatically by our specialized proof strategy, 7 require some additional interaction. However, only 4 of these 7 transformations require a non-trivial interaction. These 4 all deal with indirect loading and storing for which some additional properties have to be established. We have discovered 5 erroneous transformations in Tanenbaum's list of peephole optimization steps. In all these cases, however, we were able to identify strengthened admissibility conditions for which the optimization step is correct.

5 Concluding Remarks

We have outlined how to represent a general scheme for verifying local optimizations, how to instantiate it for a specific machine architecture, and how to encode and prove correct a set of peephole optimization rules using a specialized proof strategy. By detecting errors in published peephole optimization rules, we have demonstrated once again the importance of a rigorous formal treatment.

In order to demonstrate the wide applicability of the approach and the specialized proof strategy, we also instantiated this scheme with a formalization of a PDP-11 like two-address machine with different addressing modes [2], and proved all the optimization steps for this machine as stated in [2] to be correct [5]. In addition, we instantiated the generic scheme with the Tamarack [17] micro-processor and verified peephole optimization rules for this processor. In [5], however, we used an older version of PVS which did not provide powerful built-in proof strategies such as `GRIND`. There, the degree of automation was much lower, only 83 of 108 transformations from [14] could be proved automatically using a specific strategy.

Besides the correctness proofs of peephole optimization steps, the generic developments described in this paper can also used for establishing the correctness of other local optimization tasks like transformations to improve scheduling on RISC architectures, since these transformations can also be formalized as transformations on linear code sequences.

The generic interpreter has also been used within the Verifix project to verify local code generation rules from an intermediate language into DEC Alpha code. In addition, it has been utilized to specify and verify the compilation of standard imperative language constructs into code of an arbitrary machine [6]. To implement conditionals and while loops, jump instructions have been added to the linear code, and an interpreter for this code has been provided. Our linear interpreter presented in this paper has been embedded into this machine code interpreter. Future work will consider the verification of optimizations on code including jumps using this interpreter.

Acknowledgment

We thank all members of the Verifix team for helpful discussions on optimization techniques. The constructive criticisms and suggestions provided by the anonymous referees have greatly improved the paper.

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 and 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 and Experience*, 17(11):801–812, November 1987.

5. A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. A Generic Specification for Verifying Peephole Optimizations. Technical Report UIB-95-14, Universität Ulm, Fakultät für Informatik, 89069-Ulm, Germany, December 1995.
6. A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Generic Compilation Schemes for Simple Programming Constructs. Technical Report UIB-96-12, Universität Ulm, Fakultät für Informatik, 89069-Ulm, Germany, December 1996.
7. Andrew Gill. A Novel Approach Towards Peephole Optimisations. In *Proceedings of the 4th Annual Glasgow Workshop on Functional Programming*, Workshops in Computer Science. Springer-Verlag, August 1991.
8. Peter B. Kessler. Discovering Machine Specific Code Improvements. *Sigplan Notices*, 21(7):249–254, 1986.
9. Robert R. Kessler. Peep - an Architectural Description Driven Peephole Optimizer. *Sigplan Notices*, 19(6):106–110, June 1984.
10. David Alex Lamb. Construction of a Peephole Optimizer. *Software – Practice and Experience*, 11(6):639–647, June 1981.
11. Timothy S. McNerney. Verifying the Correctness of Compiler Transformations on Basic Blocks using Abstract Interpretation. *Sigplan Notices*, 26(9):106–115, 1991.
12. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
13. 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.
14. 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, January 1982.
15. P.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.
16. P.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.
17. P.J. Windley and M. Coe. Microprocessor Verification: A Tutorial. Technical Report LAL-92-10, University of Idaho, Department of Computer Science, Laboratory for Applied Logic, 1992.

This article was processed using the LaTeX macro package with LLNCS style