Formal Verification of Dead Code Elimination in Isabelle/HOL

Jan Olaf Blech Lars Gesellensetter Sabine Glesner

Institute for Program Structures and Data Organization
University of Karlsruhe, Karlsruhe, Germany
Email: {blech|lars}@ipd.info.uni-karlsruhe.de, glesner@ira.uka.de

Abstract

Correct compilers are a vital precondition to ensure software correctness. Optimizations are the most error-prone phases in compilers. In this paper, we formally verify dead code elimination (DCE) within the theorem prover Isabelle/HOL. DCE is a popular optimization in compilers which is typically performed on the intermediate representation. In our work, we reformulate the algorithm for DCE so that it is applicable to static single assignment (SSA) form which is a state of the art intermediate representation in modern compilers, thereby showing that DCE is significantly simpler on SSA form than on classical intermediate representations. Moreover, we formally prove our algorithm correct within the theorem prover Isabelle/HOL. Our program equivalence criterion used in this proof is based on bisimulation and, hence, captures also the case of nontermination adequately. Finally we report on our implementation of this verified DCE algorithm in the industrial-strength Scale compiler system.

1. Introduction

We address the problem of formally proving the correctness of dead code elimination (DCE) on static single assignment (SSA) intermediate representation, a state of the art intermediate language in compilers. DCE is an optimization available in almost all modern compilers which eliminates statements that are not needed by the program, thereby shortening the program representation and speeding up its execution. DCE is performed on the intermediate representation in a compiler. To formally prove the correctness of DCE on SSA form, we need to solve the following problems: First, we need to reformulate the classical DCE algorithm to be applicable to SSA form. Secondly, we have to prove the semantic equivalence of the original program and the one optimized by DCE. And finally, we need to integrate the DCE algorithm into an existing, practically used compiler infrastructure.

We require our solution to fulfill the following requirements: The DCE algorithm is to be adapted for SSA representations by exploiting their static single assignment characteristic. Also, the DCE algorithm needs to produce correct results on SSA form, i.e., it does not change the semantics of programs during their optimization. We want this correctness proof to be formulated within the Isabelle/HOL theorem prover [18]. In particular, as a prerequisite for this proof, we need a formal semantics of SSA representations. Moreover, the proof is to be based on a notion of program equivalence that captures the observable behavior of programs. On top of that, the verified algorithm is to be integrated into the Scale compiler system, an industrial-strength compiler infrastructure.

Our solution follows the usual two-fold approach of data flow analyses in compilers: First, during an analysis phase, information about the program is collected. Afterwards, this information is used to optimize the program. In our case, we perform a live variables analysis that classifies all variables in a program whether they are needed to compute results which might be part of the observable behavior of the program. In contrast to the classical live variables analysis which requires cubic time in the worst-case, we show that on SSA form it only needs quadratic time due to the information inherently available in SSA representations. Having specified this analysis phase, we state and formally verify a DCE algorithm within the theorem prover Isabelle/HOL which eliminates all variables that have not been classified as live. We show that both the original program and the one on which DCE has been performed behave equivalently. In this proof, we use the proof principle of bisimulation. Finally we report on our implementation of DCE in the Scale compiler system.

Our work contributes to the field of software engineering, especially software verification and compiler verification. Correct compilers are a necessery prerequisite to ensure software correctness since nearly all software is written

in higher programming languages and compiled into executable machine code afterwards. Our results show that important optimizations in realistic compilers can be formally verified. Moreover, our methods presented here can be applied to software transformations in general, as e.g. in software reengineering, when using design patterns, in software composition and adaptation, in model driven architectures (MDA) etc. Especially with our notion of semantic equivalence based on bisimulations, we pay attention to the situation that not only final results of computations but also the entire observable behavior of programs needs to be retained during their transformation and optimization.

This paper is structured as follows: SSA form and its advantages compared to classical intermediate languages are introduced in Section 2. In Section 3, we state a formal semantics for SSA based intermediate languages and its formalization in the theorem prover Isabelle/HOL. In Section 4, we summarize the classical algorithm for DCE and our extension of it to make it applicable to SSA form. The notion of program equivalence together with the correctness proof for our DCE algorithm, both formulated within Isabelle/HOL, can be found in Section 5. We present the implementation of DCE in the Scale system in Section 6. Finally related work is discussed in Section 7, and we give a conclusion and present ideas for future work in Section 8.

2. Static Single Assignment Form

Static single assignment (SSA) form has become the preferred intermediate representation for handling all kinds of program analyses and optimizing transformations prior to code generation [5]. Its main merits comprise the explicit representation of def-use-chains and, based on them, the ease by which further dataflow information can be derived.

By definition SSA-form requires that a program and in particular each basic block is represented as a directed graph of elementary operations (jump / branch, memory read/write, arithmetic operations on data) such that each "variable" is assigned exactly once in the program text. Only references to such variables may appear as operands in operations. Thus, an operand explicitly indicates the data dependency to its point of origin. The directed graph of an SSA-representation is an overlay of the control and data flow graph of the program. A control node may depend on a value which forces control to conditionally follow a selected path. Each basic block has one or more such control nodes as its predecessors. At entry to a block, ϕ *nodes*, $x = \phi(x_1, \ldots, x_n)$, represent the unique value assigned to variable x. This value is a selection among the values $x_1, \ldots, x_n$ where x_i represents the value of x defined on the control path through the i-th predecessor of the basic block. n is the number of predecessors of this block. Programs can easily be transformed into SSA form, cf. [14], e.g. by a tree

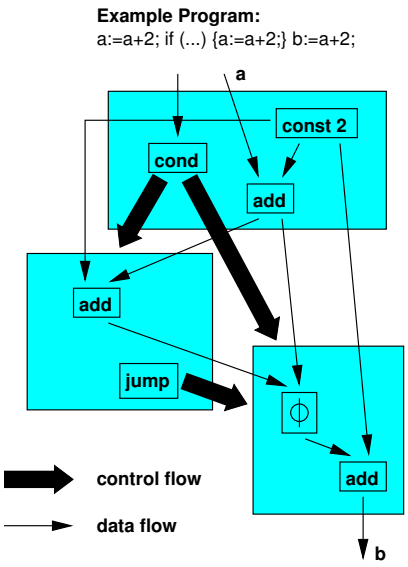

Figure 1. SSA Representation

walk through the attributed syntax tree. The standard transformation subscripts each variable. At join points, ϕ nodes sort out multiple assignments to a variable corresponding to different control flows through the program. As example, figure 1 shows the SSA form for the program fragment:

```
a := a+2; if (..) {a := a+2; } b := a+2;
```

In the first basic block, the constant 2 is added to a. The *cond* node passes control flow to the 'then' or to the 'next' *block*, depending on the result of the comparison. In the 'then' *block*, the constant 2 is added to the result of the previous *add* node. In the 'next' *block*, the ϕ node chooses which reachable definition of variable 'a' to use, the one before the if statement or the one of the 'then' *block*. The names of variables do not appear since in SSA form, variables are identified with their value and also called *value numbers*.

SSA representations describe imperative, i.e. state-based computations. A virtual machine for SSA representations starts execution with the first basic block of a given program. After execution of the current block, control flow is transferred to the then uniquely defined subsequent block. Hence, the current state is characterized by the current basic block and by the outcomes of the operations in the previously executed basic blocks.

3. SSA Formalization in Isabelle/HOL

In this section, we describe the specification of SSA based intermediate languages within the Isabelle/HOL system. The formalization can be separated into two parts, namely into the local data flow within basic blocks and the

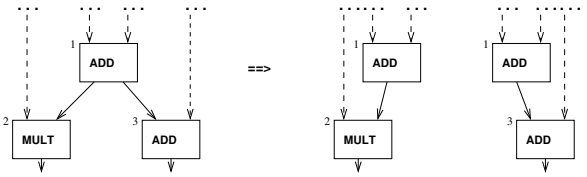

Figure 2. Transforming SSA DAGs into SSA Trees

global control and data flow which connects individual basic blocks. First, we formalize the data flow within basic blocks. Then we describe the global control and data flow. The results in this section are a summary of [2].

3.1 Data Flow within Basic Blocks

Basic blocks in SSA intermediate representations can be regarded as directed acyclic graphs (DAGs) such that the nodes represent operations (e.g. arithmetic operators, constants, or ϕ nodes) and the edges represent the data flow in-between. Evaluation of basic blocks takes place in two steps: First, the ϕ nodes are evaluated simultaneously. Then, the results of the remaining operations are determined. Therefore we can treat ϕ nodes within a given basic block as constants. Hence, constants and ϕ nodes (within a given basic block) are nodes with only outgoing edges.

DAGs representing SSA basic blocks contain common subexpressions only once. In our formalization we have represented such a DAG by transforming it into an equivalent set of trees by duplicating shared subterms, cf. Figure 2. To enable identification of equivalent subtrees, we assign a unique number to each operation in the original DAG and duplicate this identification number whenever duplicating a shared subexpression. We can transform such a set of trees into a single tree by adding a root node. In Isabelle/HOL, these trees are formalized in the following manner:

datatype $SSATree = CONST\ value\ identifier\ |$
$PHI\ phiargs\ value\ identifier\ |$
$NODE\ operator\ SSATree\ SSATree\ value\ identifier$
$\ldots$

Nodes represent constants, ϕ-nodes with argument lists and arithmetic operations. SSA basic blocks are evaluated with the evaluation function $eval\_tree$ which is defined inductively on SSA trees, thereby using the function $get\_ssatree\_val$ returning the value of the root node of a tree:

consts
$eval\_tree :: SSATree \Rightarrow SSATree$
primrec
$eval\_tree\ (CONST\ val\ ident) = (CONST\ val\ ident)$
$\ldots\ldots$
$eval\_tree\ (NODE\ operator\ tree1\ tree2\ val\ ident) =$
$(NODE\ operator\ (eval\_tree\ tree1)\ (eval\_tree\ tree2)$
$(operator\ (get\_ssatree\_val\ (eval\_tree\ tree1))$
$(get\_ssatree\_val\ (eval\_tree\ tree2)))$
$ident)\quad \ldots$

3.2 Global Control and Data Flow

An SSA program is formalized as a list of basic blocks in which each basic block carries four pieces of information which integrate it into the global control and data flow:

datatype $BASICBLOCK =$
$BB\ identifier\ ''identifier \times nat''$
$''identifier \times nat''\ ''SSATree\ list''$

1. $identifier$	the value number that determines the successor basic block
2. $identifier \times nat$	successor target 1 and its rank
3. $identifier \times nat$	successor target 2 and its rank
4. $SSATree\ list$	list of SSATrees containing the operations of the basic block

To simplify the DCE correctness proof, we represent each SSATree such that all its subtrees are also contained in the SSATree list of their respective basic block.

In our formalization, a basic block b can have two different successors b' (target 1 and target 2) specified by the second and third field of type $identifier \times nat$. $identifier$ is the number characterizing the successor block. nat specifies the rank which selects the arguments in the ϕ nodes in b': If the value of rank is i, then the ith argument in the argument list of each ϕ node in b' is chosen. (Remember that ϕ nodes have exactly as many operands as the basic block has predecessor blocks.)

Execution of SSA programs is state-based. Each single state transition corresponds to the execution of a single basic block. We define the current state by the values of the operations executed in previous basic blocks, and by the currently executed basic block. Therefore we specify a state as:

- a table of values — formalized as a function $(identifier \Rightarrow value)$ indexed by value numbers
- current basic block and its rank

The state transition function $(step :: BASICBLOCK\ list \Rightarrow state \Rightarrow state)$ evaluates basic blocks by performing the following computations:

- it assigns each ϕ-node its value
- it evaluates the basic block
  (i.e. calculates and stores values in nodes)
- it collects all calculated values and
  updates the table of values
- it determines the successor basic block depending on
  the corresponding distinct value number,
  cp. definition of the datatype $BASICBLOCK$

We have specified the semantics of SSA intermediate languages via this state transition function, thereby covering all major aspects of SSA based intermediate languages. For a complete specification with all details, we refer to [1]. Since we are working on intermediate languages without an explicit representation of exception handling, we can safely assume that possible occurences and treatments of exceptions have been replaced by if-statements in previous compiler phases. It should also be noted that our approach – looking at data dependencies when reasoning about basic blocks – is highly adequate to deal with side effects. Side effects are extra data dependencies, hence they can be dealt with as ordinary data dependencies in our representation.

4. Dead Code Elimination (DCE) on SSA Form

The task of dead code elimination (DCE) is to eliminate unused statements, i.e. assignments to variables that are not needed afterwards. In this section, we first summarize the classical approach to DCE in Subsection 4.1. Afterwards, in Subsection 4.2, we reformulate the DCE algorithm for SSA form, thereby getting a worst-case quadratic time algorithm, in contrast to the classical approach needing worst-case cubic time. In Subsection 4.3, we present our formalization of our DCE algorithm on SSA form within the theorem prover Isabelle/HOL.

4.1 Classical DCE

DCE requires a live variables analysis. Usually, a variable is defined to be live at a certain program point if there exists a path from this program point to a use of the variable that does not redefine the variable. In the classical approach to data flow analyses as described in [16], the analysis is regarded as an instance of a monotone (and even distributive) framework. That means that one constructs an equation system which describes the desired analysis result. Due to the general form of monotone frameworks, these equations do have solutions whereat typically one is interested either in the smallest or largest of them. These solutions can be computed by fixed point algorithms that need cubic time in the worst case. The theoretical basis of this approach is Tarski's fixed point theorem together with the fact that the desired solutions are fixed points of monotone functions on finite lattices.

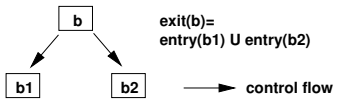

Figure 3. Live Variables Analysis

In case of the live variables analysis, the program is divided into blocks, each of them containing an individual statement. Starting from the definition that all variables that are defined in the final block of program execution are live, the program is explored in backward direction to determine which other variables are live as well. For each block, there are two sets of live variables, one characterizing the variables that are live upon exit, the other those that are live upon entry of the block. The entry-set is determined based on the exit-set. E.g. if there is an assignment `x:=a`, then the entry-set is determined by removing the variable `x` from the exit-set. Moreover, all variables contained in the expression `a` are inserted into the entry-set. Given a block b with only a single successor block b', the exit-set of b is the entry-set of b'. If b has more than one successor block, then its exit-set is the union of the entry-sets of all its successor blocks, cf. Figure 3. With these definitions, one obtains a set of equations defining the entry- and exit-sets of all blocks in the program. This set of equations has a unique smallest fixed point which characterizes all variables together with the program points at which they are live in the program. This fixed point can be computed by an iteration that needs cubic time in the worst case.

Short outline of proof that the analysis needs cubic time: Assume that n is the size of the program. Then there are $O(n)$ program points with live variables information, each information of size $O(n)$. Hence, the size of the overall analysis information is $O(n^2)$. One starts the analysis by assuming that all variables are not live and iterates by marking all variables as live which are necessary to be live in order to fulfill the equations defining the live variables analysis. Since in each step of the fixed point iteration, at least one more variable at some program point is marked live, the analysis will stop after $O(n^2)$ steps. Each step needs $O(n)$ time so that the entire analysis stops after $O(n^3)$ time.

One can use the live variables analysis information for dead code elimination: If a variable is not live at the exit of a program point, if this block is an assignment to the variable, then one can eliminate this block.

4.2. DCE on SSA

Programs in SSA form have the property that each variable is statically assigned at most once in the program. This directly implies that there are no removals from the exit- and entry-sets which significantly simplifies the analysis.

For DCE on SSA form, we use the following definition of live variables (which subsumes the classical definition presented in the previous subsection if one classifies variables in the final block as interesting): A variable is live if it is needed to calculate the values of other live variables or is classified as interesting (e.g. by being observable like input/output operations) or determines the control flow of the program. The latter is necessary because otherwise an arbitrary terminating and an arbitrary non-terminating program without output would be regarded semantically equivalent.

Compared to the classical approach of live variables analysis, described in the previous subsection, the analysis gets much simpler when operating on SSA form. Since variables in SSA form are value numbers which are assigned only once in the static program representation and the def-use chains are explicit, we only have a global set of live variables or live value numbers, resp. This set which is of size $O(n)$, n being again the program size, can be calculated via a single pass through the program representation: Initially, all program points are marked as unvisited and the variables classified as interesting as well as those determining the control flow or doing input/output operations are marked as live. Then the following step is repeated until there are no unvisited program points which are assignments to variables marked live: For each still unvisited program point that contains the assignment to a variable marked live, all variables on the right-hand side of the assignment are marked live. This algorithm needs quadratic time since in each step at least one variable is marked live. Each steps needs linear time. Since there are at most n steps, the algorithm needs quadratic time. Note that, in practice, pathological cases inducing quadratic time (or cubic time in case of the classical DCE algorithm) only rarely show up.

4.3 Formalization of DCE in Isabelle/HOL

In our formalization of DCE on SSA form, we use the formal semantics of SSA form as presented in Section 3 as basis. Formally we distinguish four sets of value numbers in our analysis:

- The set A_P contains all value numbers in a given program P.
- We provide the DCE-algorithm with a set I_P of value numbers whose values we are interested in (e.g. the input/output operations).
- We have a set C_P of control flow determining value numbers (i.e. boolean variables that appear in if-clauses).
- The set of live variables L_P contains all value numbers that may be necessary to compute C_P and I_P including transitive dependencies and C_P and I_P itself.

With this information we are able to specify DCE on programs P. The function which performs DCE,

$elim\_dead\_P :: program \Rightarrow nat\ set \Rightarrow program$

takes a program and the set of live variables for this program as determined by the analysis and returns a program on which DCE has been performed. $elim\_dead\_P$ calls

$elim\_dead :: SSATree\ list \Rightarrow identifier\ set \Rightarrow SSATree\ list$

which is defined as follows:

$elim\_dead\ []\ livevars = []$
$elim\_dead\ (tree\ \#\ trees)\ livevars =$
$(if\ get\_id\ tree \in livevars\ then$
$elim\_dead\ trees\ livevars\ else$
$(tree\ \#\ elim\_dead\ trees\ livevars))$

for each basic block. This function simply eliminates all assignments to non-live variables in a block by iterating over the SSATree list of each block and throwing out every tree whose root node does not calculate a live variable.
Note: This function is so simple because in our representation, every subtree of a tree in a tree list is contained in the tree list again.

In more detail, our DCE algorithm for SSA form carries out the following steps for a given Program P:

- Determine the set of value numbers A_P, interesting variables I_P, control-flow determining value numbers C_P and all def-use chains within a single sweep over the SSA representation.
- Calculate the set of live value numbers L_P by taking all "defs" of the reflexive, transitive closure of all def-use chains that "use" value numbers from I_P or C_P.
- Perform Dead Code Elimination by sweeping through the program representation and eliminating in each basic block all assignments to value numbers not in L_P.

Typically, the variables in the final block are contained in the set I_P of interesting variables. But more generally, interesting variables can also be other variables. In the succeeding section, we prove the correctness of this DCE algorithm.

5 Correctness of DCE on SSA Form

In this section, we present our formal proof for the correctness of dead code elimination (DCE) on SSA form within the theorem prover Isabelle/HOL. First, in Subsection 5.1, we describe our notion of program equivalence which is based on bisimulation. Then, in Subsection 5.2, we summarize our Isabelle/HOL correctness proof.

5.1. Semantical Equivalence via Bisimulation

Our principle idea is to regard two programs as semantically equivalent if they denote the same sequence of observable states.

When looking at program equivalence, we are only interested in observable states. The fact that two programs are regarded as semantically equivalent if they denote the same sequence of observable states can be reformulated in a way that the two programs have to bisimulate each other.

Definition 1 (Kripke Structures) *A Kripke structure is a five tuple (AP, S, R, S_0, L) where AP is a set of atomic propositions, S is a set of states, R is a transition relation, S_0 is the initial state and L is a labeling function mapping states to sets of atomic propositions.* $\diamond$

Hence, a Kripke structure is equivalent to an annotated state transition system.

Definition 2 (Bisimulation Relation [4]) *Let $M = (AP, S, R, S_0, L)$ and $M' = (AP, S', R', S'_0, L')$ be two Kripke structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a bisimulation relation between M and M' if and only if for all s and s', if $B(s, s')$ then the following conditions hold:*

1. $L(s) = L'(s')$
2. *For every state s_1 such that $R(s, s_1)$ there is s'_1 such that $R'(s', s'_1)$ and $B(s_1, s'_1)$*
3. *For every state s'_1 such that $R(s', s'_1)$ there is s_1 such that $R'(s, s_1)$ and $B(s_1, s'_1)$* $\diamond$

We can describe the semantics of a program as a structure M: Since the execution of a basic block is atomic, the semantics of a program is specified by a state and a state transition function. Each state consists of the number identifying the current basic block, its rank and a function mapping variables to their values. The state transition function takes such a variable mapping, the number of the current basic block, and its rank and returns the next state i.e. a variable mapping, the number of the succeeding basic block, and its rank. Kripke structures are used as follows for the specification of program semantics: The atomic propositions represent the variable value mappings, the numbers of basic blocks, and the values of the rank. S is the set of states reachable within the execution of the program M. R represents possible state transitions and the conditions under which they appear. S_0 is the initial state. L is a labeling function mapping states to their observable variable mappings, rank, basic block id. Two programs are bisimilar if there exists a bisimulation relation B such that the initial states of both programs are within the relation.

If we describe the semantics of a program as a structure M and the semantics of another program as a structure M', the bisimulation relation B is state equivalence, with an equivalence criterion that we can choose freely: E.g. we can restrict the variables that appear in the atomic propositions to live variables. Then $L(s) = L'(s')$ checks state equivalence. With the notion of bisimulation and the restriction that we only require the values of live variables to be equal in states that are in the bisimulation relation, we have a formal criterion under which two programs show the same behavior.

In our special case the requirements for a bisimulation get even simpler: We regard two programs as semantically equivalent iff:

- They start with equivalent initial states s and s'. This is denoted $s \simeq_{L_P} s'$ where L_P is the set of live variables. By equivalence we mean that the observable parts of the states must be the same, corresponding to the requirement $L(s) = L'(s')$ in Definition 2.
- For two states s and s' in the bisimulation relation, we require that the succeeding states are equivalent again. This is formalized in Isabelle/HOL as:
  $\forall\, s\ s'\ .s \simeq_{L_P} s' \longrightarrow next\ s \simeq_{L_P} next\ s'$
  where $next$ returns the succeeding state by calling the function $process\_block$ with adequate arguments.

This notion of program equivalence captures very elegantly the semantics of both terminating and nonterminating programs. With its state abstraction, it is flexible enough to prove most compiler optimizations correct. If we want to prove the correctness of a program optimization, we have to show that optimized and unoptimized programs denote the same sequence of observable states which is exactly what a bisimulation proof does.

5.2 Formal Verification of Dead Code Elimination

In this subsection, we describe our proof that our DCE algorithm on SSA form preserves the semantics of programs. This proof has been completely conducted within the theorem prover Isabelle/HOL. As stated in Section 5.1, it is sufficient to show that the original program and the transformed one (i.e. the one where dead code is eliminated) bisimulate each other. We require that both programs have the same initial states s_0 and s'_0. To show that the two programs bisimulate each other, we prove that for each two equivalent states s and s' and for every basic block b in a program P with live variable set L, the succeeding states ($process\_block$) will be equivalent as well, either if dead code elimination ($elim\_dead$) has been performed or not. The main theorem has the following form:

$\forall\ s\ s'\ b_P\ .\text{“}consistency\_assumptions\text{“} \longrightarrow$
$s \simeq_{L_P}\ s' \longrightarrow$
$process\_block\ b_P\ s$
$\simeq_{L_P}$
$process\_block\ (elim\_dead\ b_P\ L_P)\ s$

The consistency assumptions require that the program and its states are valid with respect to our SSA formalization:

1. For all trees calculating live variables, the trees that calculate arguments/intermediate results must appear somewhere in the program.
2. Each argument of a live ϕ-node must be contained in the live variables set. (Remember that we call a node or tree live if it or its root node calculate a live variable.)
3. Each subtree of a tree has to be contained in the SSATree list of a basic block. Subtrees of trees that calculate live variables have to be live again.
4. The basic blocks b_P must be basic blocks of the program P.

For example, the above second assumption that all arguments of live ϕ-nodes have to be live again can be ensured with the following predicate:

constdefs property2 ::
$SSATree\ set \Rightarrow nat\ set \Rightarrow bool$

$property2\ prog\_tree\_set\ L ==$
$\forall ident\ ident'\ pa\ val\ rank.$
$(PHI\ pa\ val\ ident) \in prog\_tree\_set \wedge$
$get\_id(PHI\ pa\ val\ ident) = ident' \wedge ident' \in L$
$\longrightarrow (pa!rank \in L)$

This predicate takes the set of all trees in a program ($prog\_tree\_set$) and the set of live variables (L) and is only true iff all live ϕ-node arguments are live again. ($l!i$ denotes the ith element in the list l.)

The proof of the main theorem requires the following lemma where the most important steps of the whole proof are performed:

lemma:
$\forall\ vt\ wt\ vt'\ wt'\ treelist.$
$\text{“}consistency\ assumptions\text{“} \wedge$
$\text{“}treelist\ in\ basic\ block\ of\ P\text{“} \wedge$
$vt \doteq_L vt' \wedge wt \doteq_L wt'$
$\longrightarrow$
$(pic\_tree\_vals\_list(eval\_tree\_list$
$(eval\_phi\_list\ treelist\ vt\ rank))\ wt)$
$\doteq_L$
$(pic\_tree\_vals\_list(eval\_tree\_list$
$(eval\_phi\_list(elim\_dead\ treelist\ L)\ vt'\ rank))\ wt')$

It states that one can evaluate the ϕ functions of a given $treelist$, evaluate the resulting trees, and finally collect the calculated values. Either if the dead code elimination has been performed or not, the collected values will be the same if they belong to the set of live variables. The $treelist$ is the main ingredient of a basic block, cf. also Section 3. The assumptions ensure that the $treelist$ is a tree list of an arbitrary basic block b_P of the given program P.

The symbol “$\doteq_L$” denotes equality of tables of values with respect to the set L of live variables. The vt, vt', wt, wt' are tables of values used to assign ϕ node values and to serve as an initial base for collecting when performing pic_tree_vals. In practice the table of values vt resp. vt' used to process ϕ node assignment and the table of values wt resp. wt' used to collect calculated values into will be the same. This lemma is more general as it allows vt resp. vt' and wt resp. wt' to be different. This general version can be verified more easily than the less general version.

The proof of this lemma is done by induction over the SSA tree list ($treelist$):

- **Base Case:** The SSA tree list is empty ($treelist = []$). The proof goal is:
  $\text{“}assumptions\text{“}$
  $\longrightarrow$
  $(pic\_tree\_vals\_list(eval\_tree\_list$
  $(eval\_phi\_list\ []\ vt\ rank))\ wt)$
  $\doteq_L$
  $(pic\_tree\_vals\_list(eval\_tree\_list$
  $(eval\_phi\_list(elim\_dead\ []\ L)\ vt'\ rank))\ wt')$
  Since one cannot eliminate trees out of an empty tree list, the following equation holds:
  $[] = (elim\_tree\_list[]L)$.
  Isabelle solves this case automatically with the predefined tactic *simp*.

- **Induction Case:** $treelist = x\#xs$:
  The proof goal is:
  $\forall\ vt\ vt'\ wt\ wt'.$
  $\text{“}assumptions\text{“}$
  $\longrightarrow$
  $(pic\_tree\_vals\_list(eval\_tree\_list$
  $(eval\_phi\_list\ xs\ vt\ rank))\ wt)$
  $\doteq_L$
  $(pic\_tree\_vals\_list(eval\_tree\_list$
  $(eval\_phi\_list(elim\_dead\ xs\ L)\ vt'\ rank))\ wt')$
  $\Longrightarrow$
  $\forall\ vt\ vt'\ wt\ wt'.$
  $\text{“}assumptions\text{“}$
  $\longrightarrow$
  $(pic\_tree\_vals\_list(eval\_tree\_list$

$$(eval\_phi\_list\ x\#xs\ vt\ rank))\ wt)$$
$$\doteq_L$$
$$(pic\_tree\_vals\_list(eval\_tree\_list$$
$$(eval\_phi\_list(elim\_dead\ x\#xs\ L)\ vt'\ rank))\ wt')$$

We make a case distinction whether the node x is

- **live:** ($x \in L$) We show that x is not eliminated by DCE (follows directly from the formalization) and that x calculates the same value in both cases. This requires another case distinction whether x is:
  - a single $CONST$ Node: trivial to prove.
  - a single PHI Node: easy to prove by using consistency assumptions.
  - a composed $NODE$ Node: we show that the value of the $NODE$ is determined only by live variables by using our consistency assumptions.
- **not live:** ($x \notin L$) we show that it does not change the overall result, no matter if x is eliminated from the $treelist$ or not.

With the help of this lemma the proof of the main theorem requires only basic proof steps and is quite easy in Isabelle/HOL.

With this proof we have ensured that our DCE algorithm preserves the semantics of programs optimized by it. In total our proof formalization within Isabelle/HOL has required 43 lemmata and theorems.

6. Implementing the Verified DCE Algorithm

For the implementation of our approach, we have employed the Scale compiler infrastructure [6] as basis. We have chosen Scale because of its well-structured conception and detailed documentation as well as because of its industrial-strength efficiency which has been demonstrated by the Scale group with published result on the efficient compilation of the complete SPEC 2000 benchmarks.

The architecture of Scale is depicted in Fig. 4. The frontends yield abstract syntax tree representations of the source files (C or Fortran code), which are in turn transformed into the internal language Scribble, representing the control flow graphs (CFG) in SSA form. On this representation, various optimizations can be performed in various orders. Eventually a selected backend translates the CFGs into machine code. Source-to-source translation is also supported.

We have realized our implementation of the verified DCE algorithm as an additional optimization. The analysis phase in Scale collects the def-use relations which are used in the transformation phase to eliminate dead code. In

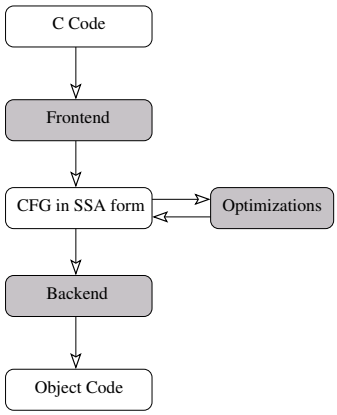

Figure 4. Overview of the Scale Compiler

Scale, the direct def-use edges between value numbers are represented explicitly in SSA form. Hence, to implement our DCE algorithm, all that we needed to do was to take the specified set of initially live value numbers (i.e. the union of I_P, the set of value numbers we are interested in, and C_P, the set of value numbers determining the control flow) and calculate the transitive closure of this set (with respect to the def-use edges), which yields the set L_P of live value numbers as result. Next, this set is used to perform the actual transformation, namely to remove all assignments to dead value numbers from the CFG.

The internal representation of Scale consists of *chord* nodes, which constitute the instructions (i.e. assigning an expression to a value number or conditional branching). They are chained to each other according to the control flow. Basic blocks are represented implicitly. The *chord* nodes receive their data from trees of *expression* nodes, e.g. literals, operators, or value numbers. For our purpose, the *assign chord* is most important as it assigns the result of an expression to a given value number which in turn directly influences the results of the live variables analysis.

Figure 5 shows the CFG after transformation to SSA form for the following simple example:

```
void main() {
  int a,b,c;
  a=20;
  b=12;
  c=14;
  if (a==10)
    c=4;
  a += c; }
```

The initial versions of the variables a, b, c are mapped to the value numbers 1, 2, and 3, respectively. The further versions of c are mapped to the value numbers 4 and 5, and the final version of a is mapped to the

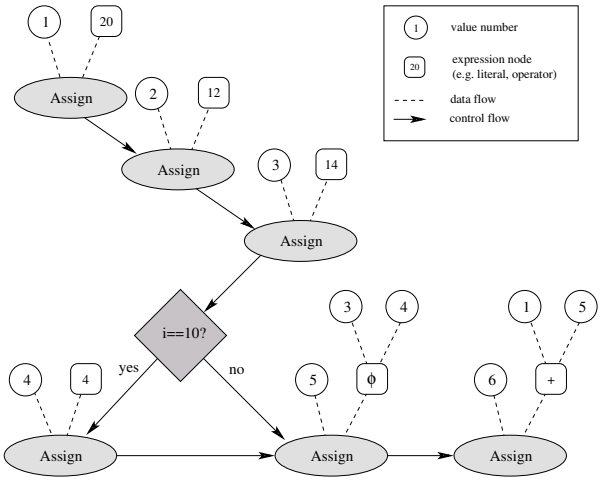

Figure 5. CFG for the example

value number 6. We regard the final value of a as an interesting value, besides the initial value of a influences the control flow. Thus we have the following sets: $A_P = \{1, 2, 3, 4, 5, 6\}, I_P = \{6\}, C_P = \{1\}$. From these sets, we get in the first iteration the set $L_P = \{1, 3, 4, 5, 6\}$. Further iteration yields the same set, which constitutes the solution. This leads to the elimination of the second assignment, since b and its value number 2, resp., are dead.

Empirical Results: As a testcase for our implementation of our verified version of Dead Code Elimination, we considered two commonly used software collections: The SPEC 2000 Benchmarks and the EDG C Frontend. The SPEC 2000 Benchmark is the state of the art corpus to check the effectiveness of a wide variety of optimizations. The choice for the EDG Frontend was motivated by the fact that it is widely used in various compiler infrastructures (among them Scale).

Corpus	Files	LOC	Value Numbers		
			Σ	Avg	Dead
SPEC 2000	474	271k	214k	39.83	0.8%
EDG Frontend	185	465k	145k	29.68	0.2%

Table 1. Empirical Results

We compiled the complete projects and collected the following information (regarding one procedure at a time): We determined how many value numbers are used (in total and on average per procedure) and how many of these value numbers are dead. As our algorithm does not deal with pointer or array datatypes and since we wanted to exactly implement this verified algorithm, we considered all values involved in pointer or array operations as live.

The results of our algorithm are shown in table 1. Even though we only optimized with a quite restricted algorithm that does not optimize any pointer or array structures nor memory accesses, we still get a noticeable improvement of 0.8% and 0.2%, resp., of code that can safely be eliminated. In the experiments, no additional optimizations were performed. With these results, we have shown that verified optimization algorithms can be integrated into real-life compiler systems and that they can effectively improve the translated and optimized programs.

7. Related Work

The area of compiler validation and verification is an active field of ongoing research. Typically one distinguishes two notions of compiler correctness: the correctness of compilation algorithms and the correctness of compiler implementations. The first notion, the correctness of compilation algorithms, deals with the question whether a given algorithm preserves the semantics of the programs during their compilation. The second notion considers the question if a given compilation algorithm is correctly implemented.

The first work on the correctness of compilation algorithms is [12] considering the correctness of a very simple compiler for arithmetic expressions. Early work on compiler verification with the help of theorem provers is described in [13]. Recent work has concentrated on transformations taking place in compiler frontends. [17] describes the verification of the lexical analysis in Isabelle/HOL. The formal verification of the translation from Java to Java byte code and formal byte code verification was investigated in [21, 11]. The german Verifix project [9] which was funded by the German Science Foundation (DFG) developed methods to construct correct compilers which are as efficient as typical commercial compilers. [7] describes Verifix results and considers the verification of a compiler for a Lisp subset in the theorem prover PVS.

The question of implementation correctness was investigated in [3]. The Verifix project proposed the method of *program checking* [10] which was independently investigated in [19] as *translation validation*. The idea is to not validate/verify the compiler implementation itself (which would be much too expensive) but to only validate/verify the compiler result. In many cases, this check is much easier. The approach of proof-carrying code [15] is weaker than ours because it concentrates only on the verification of necessary but not sufficient correctness criteria. A preversion of the proof presented in this paper is described in [8] which is again based on the work published in [1, 2].

There has also been research investigating the connections between various forms of program analysis, e.g. the work showing that data flow analysis is model checking of

abstract interpretations [20]. In contrast to our approach, this only considers abstractions of the program behavior and, hence, only proves certain aspects of the semantics.

Concerning the validation and verification of program optimizations (such as dead code elimination), the checker approach seems to be particularly promising. Typically, such an optimization requires an analysis beforehand that determines the program parts which can safely be optimized. In many cases, the analysis information can be verified much more easily than computed. It is subject of our future work to implement a checker for the live variables analysis as well as to investigate and verify further compiler optimizations.

8. Conclusion and Future Work

We have formally specified and verified a well-known compiler optimization algorithm, namely dead code elimination, within the theorem prover Isabelle/HOL. Dead code elimination (DCE) is performed in almost all of today's optimizing compilers. Based on this work, we have implemented the DCE algorithm for the Scale compiler infrastructure, yielding noticeable optimization results. Our work shows that formal specification and verification of real-life algorithms in theorem provers are possible and that the verified algorithm can be implemented in a real-life compiler system.

In future work we want to investigate further optimizations. In particular we want to exploit the given situation that most optimizations require a preceding analysis whose results are used subsequently to optimize programs. It seems that such analysis results can be checked much more easily than computed, thus preparing the ground for the systematic use of program checking in order to make sure that not only the compiler algorithm but also its implementation behaves correctly. We are planning to investigate if such optimizing transformations and their checkers can be automatically generated from a formal specification. This would not only be important for the field of compiler verification but also for the verification of general software transformations (as e.g. in software reengineering, when using design patterns, in software composition and adaptation, in model driven architectures (MDA) etc.) since these transformations work according to the same principles as the data flow analyses one of them has been investigated in this paper.

Acknowledgement: Many thanks to Denise Dudek for helpful discussions and cooperation on this research.

References

[1] J. O. Blech. Eine formale Semantik für SSA-Zwischensprachen in Isabelle/HOL. Diplomarbeit (Master's Thesis), Universität Karlsruhe, 2004.

[2] J. O. Blech and S. Glesner. A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In *Proc. 3. Arbeitstagung Programmiersprachen (ATPS),*. LNI, September 2004.

[3] L. M. Chirica and D. F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, 1986.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[6] Scale Compiler Group, Department of Computer Science, University of Massachusetts. Scale Home Page, 2005. http://www-ali.cs.umass.edu/Scale/.

[7] A. Dold, F. W. von Henke, and W. Goerigk. A Completely Verified Realistic Bootstrap Compiler. *International Journal of Foundations of Computer Science*, 14(4):659–680, 2003.

[8] D. Dudek. Maschinelle Verifikation der Eliminierung toten Codes in SSA-Darstellungen. Studienarbeit (Minor Thesis), Universität Karlsruhe, 2005.

[9] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46:265–276, 2004.

[10] A. Heberle, T. Gaul, W. Goerigk, G. Goos, and W. Zimmermann. Construction of Verified Compiler Front-Ends with Program-Checking. In *Perspectives of System Informatics, PSI'99*, 1999. Springer LNCS Vol. 1755.

[11] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[12] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proc. Symposia in Applied Mathematics*, American Mathematical Society, 1967.

[13] J. S. Moore. A Mechanically Verified Language Implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.

[14] S. S. Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[15] G. C. Necula. Proof-Carrying Code. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[17] T. Nipkow. Verified Lexical Analysis. In *Theorem Proving in Higher Order Logics*. Springer LNCS Vol. 1479, 1998.

[18] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Lecture Notes in Computer Science, Vol. 2283, 2002.

[19] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, 1998. Springer LNCS Vol. 1384.

[20] D. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In *Proc. 5th Static Analysis Symposium*. Springer LNCS 1503, 1998.

[21] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*. Springer LNCS Vol. 2392, 2002.