

Verified Dead Store Elimination via Backwards Dataflow Analysis in Lean 4

Abstract

We describe a mechanised development in Lean 4 of dead store elimination for a small imperative language with assignments, sequencing, conditionals and while loops. The optimiser is driven by a backwards liveness analysis whose loop case is the least fixed point of a monotone functional on a finite powerset, computed by an executable Kleene iteration with a proved termination measure. The central theorem states that the optimised program is observationally equivalent to the source with respect to a chosen set of observed variables: each terminates exactly when the other does, and terminating runs end in states that agree on the observed variables. The theorem is proved twice, once by induction over big-step derivations in both directions and once by a stuttering forward simulation on the small-step semantics combined with a well-founded progress argument and determinism. The development uses only the Lean core library, contains no admitted goals or project axioms, and its main theorems depend only on the standard axioms of propositional extensionality, choice and quotient soundness.

1 Introduction

Dead store elimination discards assignments whose written values are never read before the variable is assigned again or the program ends. It is a small optimisation, but its basis is the one shared by every dataflow-driven transformation: the compiler solves equations recursively over the program text to obtain a syntactic approximation of a semantic property, and the transformation is correct only if that approximation really bounds the property along every possible path through the program's execution. For liveness the equations are solved backwards through the program, while its interpretation moves forwards from an initial state through successive configurations. A verified optimiser therefore has to connect a backwards syntactic computation with a forwards semantic one, and that connection must hold for loops, where the backwards equation is recursive and only acquires a meaning through a fixed point.

CompCert shows that such connections can be made at the level of a practically viable compiler [7, 8, 9], and Bertot, Grégoire and Leroy showed how the proof of a dataflow-based optimisation can be organised around a general solver together with a specific soundness proof for each transformation [2]. Benton proves dead code elimination correct in a relational setting by proposing an agreement relation, indexed by the live variables, that relates the executions of the two programs [1]. The textbook of Nipkow and Klein carries out liveness-based dead assignment elimination for a comparable language in Isabelle/HOL [10]. The current work belongs to this series of results and does not claim a novel approach. Its intention is to formalise the whole argument in Lean 4 [4] at a size that remains manageable when read in its entirety, while the parts that are easy to simplify away keep their integrity: termination of the fixed-point solver without an iteration bound, divergence preservation without a fuel parameter, and a simulation argument that states exactly what is needed beyond the forward direction alone.

The contributions of the artefact are the following. A small-step operational semantics is presented in which termination and divergence are defined without any fuel argument, and determinism, progress, uniqueness of outcomes, the equivalence of two formulations of divergence

and the equivalence with a big-step relation are proved (Section 2). The equations defining backwards liveness are given together with their monotonicity and an upper bound expressed through the program's variables (Section 4). A terminating iteration over the finite powerset of the program's variables is presented and shown to produce the least fixed point (Section 5). Dead store elimination is defined, and it is proved that it introduces no additional variables, that it respects set equality of live sets and that it is idempotent (Section 6). Observational equivalence of source and target is proved by induction on big-step derivations and, independently, as a stuttering simulation (Section 7). Section 8 describes the mechanisation together with its trusted base, and Section 9 discusses the limitations.

Throughout this paper the names of Lean declarations are given in typewriter font to aid checking the paper against the sources. Every claim made here is supported by a proof contained in the artefact, and every proof sketch follows the structure of the mechanised proof.

2 Language and operational semantics

2.1 Syntax and states

Expressions and statements are the inductive types

$$\begin{aligned} e \in \text{Expr} &::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 < e_2 \mid e_1 = e_2 \\ s \in \text{Stmt} &::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s \end{aligned}$$

where n ranges over the integers and x over strings (`Liveness.Expr`, `Liveness.Stmt`). A state $\sigma \in \text{State}$ is a total function from strings to integers, and $\sigma[x \mapsto v]$ is the update that returns v at x and $\sigma(y)$ at $y \neq x$ (`State.update`, with the lookup lemmas `update_same` and `update_other`). Expression evaluation $\llbracket e \rrbracket_\sigma$ is a total function defined by structural recursion (`eval`), and a comparison evaluates to 1 when it holds and to 0 otherwise. Expressions have no side effects and cannot fail, so there is no exception or error state anywhere in the language.

Conditionals and loops evaluate the integer 0 as false and any nonzero integer as true. This is the only adherence to a truth encoding in the development, and it appears only in the inference rules for conditionals.

2.2 Small-step semantics

A configuration is a pair (s, σ) of a statement and a state (`Config`). The step relation $\rightarrow$ is the inductive predicate `Step` with the rules

$$\begin{array}{c} \frac{}{(x := e, \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto \llbracket e \rrbracket_\sigma])} \quad \frac{}{(\text{skip}; s, \sigma) \rightarrow (s, \sigma)} \quad \frac{(s_1, \sigma) \rightarrow (s'_1, \sigma')}{(s_1; s_2, \sigma) \rightarrow (s'_1; s_2, \sigma')} \\[10pt] \frac{\llbracket e \rrbracket_\sigma \neq 0}{(\text{if } e \text{ then } s_1 \text{ else } s_2, \sigma) \rightarrow (s_1, \sigma)} \quad \frac{\llbracket e \rrbracket_\sigma = 0}{(\text{if } e \text{ then } s_1 \text{ else } s_2, \sigma) \rightarrow (s_2, \sigma)} \\[10pt] \frac{}{(\text{while } e \text{ do } s, \sigma) \rightarrow (\text{if } e \text{ then } (s; \text{while } e \text{ do } s) \text{ else skip}, \sigma)} \end{array}$$

Its reflexive transitive closure $\rightarrow^*$ (`Steps`) and the step-indexed relation $\rightarrow^n$ (`StepsN`) are defined as inductive predicates, since the Lean core library used by the development does not provide a reflexive transitive closure. The closure satisfies the expected lemmas: it is transitive, it is congruent for the first component of a sequence (`Steps.seq_congr`), and $c \rightarrow^* c'$ holds exactly when $c \rightarrow^n c'$ for some n .

Definition 2.1 (Termination and divergence).

$$\text{Terminates}(s, \sigma, \sigma') :\Leftrightarrow (s, \sigma) \rightarrow^* (\text{skip}, \sigma')$$

$$\text{Diverges}(c) :\Leftrightarrow \forall n. \exists c'. c \rightarrow^n c'$$

$$\text{InfiniteTrace}(c) :\Leftrightarrow \exists f : \mathbb{N} \rightarrow \text{Config}. f(0) = c \wedge \forall n. f(n) \rightarrow f(n+1)$$

Fuel is irrelevant to both the concept of termination and that of divergence. Divergence is described as having runs of every finite length; an infinite run is shown below to be equivalent and is therefore a consequence rather than a separate definition.

Theorem 2.2 (Basic properties, Liveness/Semantics.lean).

1. *The step relation is deterministic: if $c \rightarrow c_1$ and $c \rightarrow c_2$ then $c_1 = c_2$ (**Step.deterministic**). No step leaves (skip, σ) (**skip_stuck**).*
2. *Every configuration whose statement is not **skip** can step (**progress**).*
3. *Terminating results are unique: if $\text{Terminates}(s, \sigma, \sigma_1)$ and $\text{Terminates}(s, \sigma, \sigma_2)$ then $\sigma_1 = \sigma_2$ (**Terminates.unique**).*
4. *$\text{Diverges}(s, \sigma)$ holds if and only if there is no σ' with $\text{Terminates}(s, \sigma, \sigma')$ (**diverges_iff_not_terminates**); consequently every configuration either terminates or diverges (**terminates_or_diverges**).*
5. *$\text{Diverges}(c)$ holds if and only if $\text{InfiniteTrace}(c)$ (**diverges_iff_infiniteTrace**).*

Proof sketch. Determinism is an induction on the first derivation with inversion of the second; the only cases needing an argument are the two sequencing rules, which are separated by the fact that **skip** is stuck, and the two conditional rules, separated by the guard value. Progress is a structural induction on the statement. Uniqueness is an induction on the first execution: at each step the second execution cannot have stopped, since **skip** is stuck, so its first step coincides by determinism. For item 4, if c diverges and terminates in k steps then an execution of length $k+1$ exists and by step-indexed determinism its first k steps reach (skip, σ') , which cannot step. Conversely, if c does not terminate then by induction on n an execution of length n ending in a configuration that is not **skip** can be extended by progress. Item 5 from left to right uses classical choice to select, for each n , the configuration at distance n ; step-indexed determinism shows that consecutive selections are related by a step. The converse direction is a direct induction. $\square$

2.3 Big-step semantics

The development also defines a big-step relation $\sigma \Downarrow_s \sigma'$ (**BigStep**) with the usual seven rules, including the two loop rules

$$\frac{\llbracket e \rrbracket_\sigma = 0}{\sigma \Downarrow_{\text{while } e \text{ do } s} \sigma} \qquad \frac{\llbracket e \rrbracket_\sigma \neq 0 \quad \sigma \Downarrow_s \sigma' \quad \sigma' \Downarrow_{\text{while } e \text{ do } s} \sigma''}{\sigma \Downarrow_{\text{while } e \text{ do } s} \sigma''}$$

Theorem 2.3 (**bigStep_iff_terminates**). $\sigma \Downarrow_s \sigma'$ holds if and only if $\text{Terminates}(s, \sigma, \sigma')$.

Proof sketch. From big-step to small-step, induction on the derivation using transitivity and the sequencing congruence. From small-step to big-step, the key lemma **BigStep.of_step** states that if $(s, \sigma) \rightarrow (s', \sigma')$ and $\sigma' \Downarrow_{s'} \sigma''$ then $\sigma \Downarrow_s \sigma''$; it is an induction on the step with inversion of the big-step derivation, and the loop unfolding case reassembles the two loop rules. The result follows by induction on the length of the execution. $\square$

The big-step relation is a proof tool. The correctness theorem is stated using the small-step notions of Definition 2.1, and the big-step relation is used in one of the two proofs because induction on derivations gives immediate access to the loop invariant.

2.4 Executable semantics

The step function $stepFn : Stmt \rightarrow State \rightarrow Option Config$ gives the unique successor of a configuration, and it has been proved both sound and complete with respect to $\rightarrow$ (`stepFn_iff`). The runner run , which applies it as long as fuel remains, has been proved to report only real terminating executions (`run_finished_sound`) and to report every terminating execution once the fuel reaches the length of its step sequence (`run_complete`), so a program terminates from σ exactly when some fuel makes the runner finish (`run_finished_iff`), and a diverging program exhausts every fuel (`run_outOfFuel_of_diverges`). These functions are used to run the examples in the test executable; they contribute to no theorem about the optimiser.

3 Observations and free variables

Definition 3.1 (Agreement and observational equivalence). For a finite set L of variables, $Agree(L, \sigma_1, \sigma_2)$ holds when $\sigma_1(x) = \sigma_2(x)$ for every $x \in L$. Statements s and t are observationally equivalent with respect to L , written $ObsEquiv(L, s, t)$, when for every initial state σ ,

$$\begin{aligned} \forall \sigma_1. \text{Terminates}(s, \sigma, \sigma_1) &\Rightarrow \exists \sigma_2. \text{Terminates}(t, \sigma, \sigma_2) \wedge Agree(L, \sigma_1, \sigma_2) \quad \text{and} \\ \forall \sigma_2. \text{Terminates}(t, \sigma, \sigma_2) &\Rightarrow \exists \sigma_1. \text{Terminates}(s, \sigma, \sigma_1) \wedge Agree(L, \sigma_1, \sigma_2). \end{aligned}$$

Agreement is an equivalence relation on states for each L , is antitone in L and is compatible with union (`Agree.mono`, `Agree.union`). Two lemmas about updates carry the assignment cases of every later proof: agreement on $L \setminus \{x\}$ is preserved by updating x to the same value on both sides (`Agree.update_same_value`), and agreement on L is preserved by updating a variable outside L on one side only (`Agree.update_dead_left`).

Observational equivalence has the properties of reflexivity, symmetry and transitivity, and it is antitone with respect to L (`ObsEquiv.refl`, `ObsEquiv.symm`, `ObsEquiv.trans`, `ObsEquiv.mono`). Because both directions require termination, equivalent programs diverge from precisely the same initial states (`ObsEquiv.diverges_iff`), which follows from Theorem 2.2(4). Agreement on L provides no information about the variables outside L , which is what makes it applicable to dead store elimination, a transformation that does change the final values of unobserved variables.

Free variables $FV(e)$ of an expression are collected by structural recursion (`freeVars`), and $vars(s)$ collects every variable a statement mentions (`Stmt.vars`).

Lemma 3.2 (Dependency, `eval_agree`). *If $Agree(FV(e), \sigma_1, \sigma_2)$ then $\llbracket e \rrbracket_{\sigma_1} = \llbracket e \rrbracket_{\sigma_2}$.*

This is the single semantic principle about expressions used throughout the rest of the development. It is applied through the corollary `eval_agree_of_subset`: agreement on any superset of $FV(e)$ is all it takes.

3.1 Representation of variable sets

The development does not depend on Mathlib. Finite sets of variables are represented by lists of strings (`VarSet`) with an insertion-based union and a filter-based removal, and every property is stated through membership and the list subset relation $\subseteq$. Two lists are set-equal, written $A \equiv B$, when $A \subseteq B$ and $B \subseteq A$ (`VarSet.Equiv`). All results about live sets hold up to this equivalence, and the optimiser is proved to produce syntactically equal output on set-equal inputs (Section 6). The representation is adequate for the finite-height argument of Section 5 because the measure counts elements of a fixed universe list that are not yet members of the iterate.

4 Backwards liveness analysis

The analysis computes, for a statement s and a set L of variables that are live after s , the set $\text{liveIn}(s, L)$ of variables live before s : those whose values before s may determine the values of L after s . The transfer equations are

$$\begin{aligned} \text{liveIn}(\text{skip}, L) &= L \\ \text{liveIn}(x := e, L) &= \begin{cases} (L \setminus \{x\}) \cup FV(e) & x \in L \\ L & x \notin L \end{cases} \\ \text{liveIn}(s_1; s_2, L) &= \text{liveIn}(s_1, \text{liveIn}(s_2, L)) \\ \text{liveIn}(\text{if } e \text{ then } s_1 \text{ else } s_2, L) &= FV(e) \cup \text{liveIn}(s_1, L) \cup \text{liveIn}(s_2, L) \\ \text{liveIn}(\text{while } e \text{ do } s, L) &= \mu X. L \cup FV(e) \cup \text{liveIn}(s, X) \end{aligned}$$

In Lean these are the function `liveIn` together with one rewrite lemma per case (`liveIn_skip`, `liveIn_assign_live`, `liveIn_assign_dead`, `liveIn_seq`, `liveIn_ite`, `liveIn_while`). The loop case is the least fixed point of the loop functional $F_{e,s,L}(X) = L \cup FV(e) \cup \text{liveIn}(s, X)$ (`loopFunctional`) computed by the solver of Section 5 over the universe $U = L \cup FV(e) \cup \text{vars}(s)$.

The definition of liveIn and the proof that it stays inside its universe cannot be separated. The solver takes as an argument a proof that the functional maps subsets of U to subsets of U , and for the loop functional that proof needs the bound $\text{liveIn}(s, X) \subseteq X \cup \text{vars}(s)$ for the loop body, which is an instance of the property being established for liveIn itself. The mechanisation therefore defines a function `liveInB` returning a dependent pair of a set and a proof of the bound $\text{liveIn}(s, L) \subseteq L \cup \text{vars}(s)$, by structural recursion on s , and defines `liveIn` as its first projection. The bound is then available as `liveIn_subset` and the loop case unfolds by `liveIn_while`, which holds by reflexivity because the proof component is irrelevant.

Theorem 4.1 (Properties of the analysis, `Liveness/Analysis/Transfer.lean`).

1. $\text{liveIn}(s, \cdot)$ is monotone: $L_1 \subseteq L_2$ implies $\text{liveIn}(s, L_1) \subseteq \text{liveIn}(s, L_2)$ (`liveIn_mono`); hence it respects set equality (`liveIn_equiv`).
2. $\text{liveIn}(s, L) \subseteq L \cup \text{vars}(s)$ (`liveIn_subset`).
3. The loop functional $F_{e,s,L}$ is monotone (`loopFunctional_mono`) and bounded by U (`loopFunctional_bounded`).
4. For $X = \text{liveIn}(\text{while } e \text{ do } s, L)$: $L \cup FV(e) \cup \text{liveIn}(s, X) \subseteq X$ (`liveIn_while_postfixed`), $X \equiv L \cup FV(e) \cup \text{liveIn}(s, X)$ (`liveIn_while_fixed`), and $X \subseteq Y$ for every Y with $L \cup FV(e) \cup \text{liveIn}(s, Y) \subseteq Y$ (`liveIn_while_least`). The three projections $L \subseteq X$, $FV(e) \subseteq X$ and $\text{liveIn}(s, X) \subseteq X$ are the facts the correctness proofs use (`liveIn_while_out`, `liveIn_while_cond`, `liveIn_while_body`).

Proof sketch. Monotonicity is a structural induction on s . The assignment case splits on membership of x in each set; the one nontrivial subcase is $x \notin L_1$ and $x \in L_2$, where $L_1 \subseteq (L_2 \setminus \{x\}) \cup FV(e)$ because every element of L_1 differs from x . The loop case uses monotonicity of the least fixed point operator in its functional (Theorem 5.2(5)) together with the induction hypothesis for the body. Item 4 is the specialisation of Theorem 5.2 to the loop functional. $\square$

Theorem 4.2 (Soundness, `liveness_sound`). If $\text{Agree}(\text{liveIn}(s, L), \sigma_1, \sigma_2)$, $\sigma_1 \Downarrow_s \sigma'_1$ and $\sigma_2 \Downarrow_s \sigma'_2$, then $\text{Agree}(L, \sigma'_1, \sigma'_2)$.

Proof sketch. Induction on the first derivation, generalising L and the second run. For a live assignment the right-hand sides evaluate equally by Lemma 3.2 because $FV(e) \subseteq \text{liveIn}$, and agreement on $L \setminus \{x\}$ lifts through the update. For a dead assignment both updates are to a variable outside L . Sequencing threads $\text{liveIn}(s_2, L)$ as the intermediate set. Conditionals evaluate the guard equally because $FV(e) \subseteq \text{liveIn}$, so both derivations use the same branch. For a loop with fixed point $X = \text{liveIn}(\text{while } e \text{ do } s, L)$, the hypothesis is agreement on X ; the guards agree because $FV(e) \subseteq X$; the body hypothesis applies with live-out X because $\text{liveIn}(s, X) \subseteq X$ and yields agreement on X after the body; and the loop hypothesis, applied with the same L , closes the case. In the exit case agreement on L follows from $L \subseteq X$. $\square$

Theorem 4.2 is the formal statement behind the phrase that liveness is a backwards approximation of a forwards dependency. It does not mention the optimiser; Section 7 applies the same proof structure to the correspondence between a program and its optimised version.

5 Finite fixed-point computation

Let U be a list of variables and F a function on variable sets that is bounded by U , meaning $X \subseteq U$ implies $F(X) \subseteq U$ (**Bounded**). A set X is a post-fixed point of F when $F(X) \subseteq X$ (**PostFixed**). The solver is the function

$$\text{iterate}(U, F, X) = \begin{cases} X & F(X) \subseteq X \\ \text{iterate}(U, F, X \cup F(X)) & \text{otherwise} \end{cases} \quad \text{lfp}(U, F) = \text{iterate}(U, F, \emptyset)$$

where the Lean function `iterate` additionally takes the proof that F is bounded by U and a proof that $X \subseteq U$, and the recursive call supplies the proof that $X \cup F(X) \subseteq U$. The test $F(X) \subseteq X$ is decidable for lists of strings, so the function is executable.

Theorem 5.1 (Termination). *The recursion terminates. The measure is the number of elements of U that are not members of X , written $\text{deficit}(U, X)$ (`VarSet.deficit`). When $F(X) \not\subseteq X$ there is some $v \in F(X)$ with $v \notin X$; since $F(X) \subseteq U$, v is an element of U that is in $X \cup F(X)$ but not in X , and since $X \subseteq X \cup F(X)$ every element of U outside $X \cup F(X)$ is outside X , so the deficit strictly decreases (`VarSet.deficit_lt`).*

Lean’s termination checker accepts the well-founded recursion with this measure, and the proof that the measure decreases is given explicitly in the definition. There is no limit on iterations, and there is no branch that produces an unverified output once a limit is exceeded: the function is total because its preconditions guarantee that the measure decreases.

Theorem 5.2 (Fixed point, Liveness/Analysis/FixedPoint.lean). *Let F be bounded by U .*

1. $F(\text{lfp}(U, F)) \subseteq \text{lfp}(U, F)$ (`lfp_postfixed`).
2. $\text{lfp}(U, F) \subseteq U$ (`lfp_subset_universe`).
3. If F is monotone then $\text{lfp}(U, F) \subseteq Y$ for every post-fixed point Y of F (`lfp_least`).
4. If F is monotone then $\text{lfp}(U, F) \subseteq F(\text{lfp}(U, F))$, so $F(\text{lfp}(U, F)) \equiv \text{lfp}(U, F)$ (`lfp_prefixixed`, `lfp_fixed`).
5. If F is monotone and $F(X) \subseteq F'(X)$ for all X , where F' is bounded by some U' , then $\text{lfp}(U, F) \subseteq \text{lfp}(U', F')$ (`lfp_mono`).

Proof sketch. Items 1 and 2 are proved for every starting set $X \subseteq U$ by the functional induction principle generated for `iterate`: in the base case $F(X) \subseteq X$ holds by the test, and in the step case the induction hypothesis applies to $X \cup F(X)$. Item 3 is proved in the generalised form that

if every post-fixed point Y contains the starting set X then every post-fixed point contains the result; the step case shows $X \cup F(X) \subseteq Y$ from $X \subseteq Y$, monotonicity and $F(Y) \subseteq Y$. Item 4 applies item 3 to $Y = F(\text{lfp})$, which is post-fixed by monotonicity applied to item 1. Item 5 applies item 3 to $Y = \text{lfp}(U', F')$, which is post-fixed for F because $F(Y) \subseteq F'(Y) \subseteq Y$. $\square$

The result is the least fixed point in the sense of Tarski [11] restricted to the finite lattice of subsets of U , obtained by the ascending Kleene iteration that underlies iterative dataflow analysis [6, 5] and abstract interpretation [3]. The iteration starts from the empty set and adds $F(X)$ at each step. For a monotone F the iterates are exactly $F^n(\emptyset)$, but the proof does not need this; it needs only that the result is post-fixed, lies below every post-fixed point and lies inside U . Leastness is what makes the analysis precise rather than merely sound: any post-fixed point would make the correctness theorem go through, but a larger one would retain more stores. In particular the whole universe U is a post-fixed point, and choosing it would turn the loop case into a transformation that eliminates nothing inside loops.

6 Dead store elimination

The transformation is the function $dse : \text{Stmt} \rightarrow \mathcal{P}(\text{Var}) \rightarrow \text{Stmt} \times \mathcal{P}(\text{Var})$ that returns the optimised statement together with its live-in set:

$$\begin{aligned}
dse(\text{skip}, L) &= (\text{skip}, L) \\
dse(x := e, L) &= \begin{cases} (x := e, (L \setminus \{x\}) \cup FV(e)) & x \in L \\ (\text{skip}, L) & x \notin L \end{cases} \\
dse(s_1; s_2, L) &= (t_1; t_2, L_1) \\
&\quad \text{where } (t_2, L_2) = dse(s_2, L) \text{ and } (t_1, L_1) = dse(s_1, L_2) \\
dse(\text{if } e \text{ then } s_1 \text{ else } s_2, L) &= (\text{if } e \text{ then } t_1 \text{ else } t_2, FV(e) \cup L_1 \cup L_2) \\
&\quad \text{where } (t_1, L_1) = dse(s_1, L) \text{ and } (t_2, L_2) = dse(s_2, L) \\
dse(\text{while } e \text{ do } s, L) &= (\text{while } e \text{ do } t, X) \\
&\quad \text{where } X = \text{liveIn}(\text{while } e \text{ do } s, L) \text{ and } (t, _) = dse(s, X)
\end{aligned}$$

We write $\text{optimise}(s, L)$ for the first component (`optimise`). The second component is proved equal to $\text{liveIn}(s, L)$ (`dse_liveIn`), so the transformation and the analysis agree at every point, and the equations for optimise are available as rewrite lemmas (`optimise_skip`, `optimise_assign_live`, `optimise_assign_dead`, `optimise_seq`, `optimise_ite`, `optimise_while`). In the loop case the body is optimised against the fixed point X , which is the set live at the head of the loop on every iteration.

Theorem 6.1 (Structural properties, Liveness/Optimisation/DSE.lean).

1. $\text{vars}(\text{optimise}(s, L)) \subseteq \text{vars}(s)$ (`optimise_vars_subset`).
2. The size of $\text{optimise}(s, L)$ is at most the size of s (`optimise_size_le`).
3. If $L_1 \equiv L_2$ then $\text{optimise}(s, L_1) = \text{optimise}(s, L_2)$ (`optimise_congr`).

Determinism of the transformation is immediate because it is a total function; item 3 gives the exact formulation of that statement for the list representation, saying that the output is determined only by the set denoted by the input.

Eliminated stores leave `skip` residue inside sequences. The function `compact` (`Liveness/Optimisation/Compact`) folds `skip`; t and t ; `skip` to t , recursively through branches and loop bodies, and $\text{optimiseCompact}(s, L) = \text{compact}(\text{optimise}(s, L))$. Compaction preserves big-step derivations in both directions

(`compact_bigStep_iff`), the loop case by an induction on the derivation that replaces the body with any statement admitting the same derivations, so it is observationally equivalent to its input on every observation set (`compact_obsEquiv`), and the composed pass is correct by transitivity (`optimiseCompact_correct`) and inherits the bounds on variables and size.

Theorem 6.2 (Idempotence, Liveness/Optimisation/Idempotence.lean). *For all $Y \subseteq X$, $\text{liveIn}(\text{optimise}(s, X), Y) \equiv \text{liveIn}(s, Y)$ (`liveIn_optimise`) and $\text{optimise}(\text{optimise}(s, X), Y) = \text{optimise}(s, Y)$ (`optimise_optimise`). In particular $\text{optimise}(\text{optimise}(s, L), L) = \text{optimise}(s, L)$ (`optimise_idem`).*

Proof sketch. Both statements are structural inductions on s , the second using the first. The side condition $Y \subseteq X$ is needed: an assignment removed as dead for X is still dead for the smaller Y , but could be live for a larger set. In the sequencing case the induction hypothesis for s_1 is instantiated at $\text{liveIn}(s_2, X)$ and $\text{liveIn}(s_2, Y)$, which are in the subset relation by monotonicity. In the loop case let $\hat{X} = \text{liveIn}(\text{while } e \text{ do } s, X)$ and $\hat{Y} = \text{liveIn}(\text{while } e \text{ do } s, Y)$, so $\hat{Y} \subseteq \hat{X}$ by monotonicity, and let $t = \text{optimise}(s, \hat{X})$. The fixed point Y' for the loop $\text{while } e \text{ do } t$ at Y is below $\hat{Y}$ because $\hat{Y}$ is post-fixed for the functional of t : by the induction hypothesis at $\hat{Y} \subseteq \hat{X}$, $\text{liveIn}(t, \hat{Y}) \equiv \text{liveIn}(s, \hat{Y}) \subseteq \hat{Y}$. Conversely $Y' \subseteq \hat{X}$, so the induction hypothesis at Y' gives $\text{liveIn}(s, Y') \equiv \text{liveIn}(t, Y') \subseteq Y'$, making Y' post-fixed for the functional of s , and leastness gives $\hat{Y} \subseteq Y'$. The statement about `optimise` then follows by rewriting the live set with `optimise_congr`. $\square$

7 Semantic preservation

Theorem 7.1 (Correctness, `dse_correct`). *For every statement s and observation set L , $\text{ObsEquiv}(L, s, \text{optimise}(s, L))$.*

By the properties of `ObsEquiv` this yields the corollaries that s and $\text{optimise}(s, L)$ terminate from the same initial states (`dse_preserves_termination`), diverge from the same initial states (`dse_preserves_divergence`), and, when both terminate from a common initial state, end in states agreeing on L (`dse_final_agree`).

Both proofs of Theorem 7.1 can be found in the artefact. The first, in `Liveness/Optimisation/Correctness.lean`, goes through the big-step relation and proceeds constructor by constructor. The second, in `Liveness/Simulation/`, uses the small-step semantics and separates out the pieces needed to keep divergence across the transformation.

7.1 Proof by induction on derivations

The theorem is proved as a corollary of a proposition in which the two initial states agree on the live-in set, rather than being the same state. This modification is needed because, when the proof proceeds through a sequence $s_1; s_2$, the states after s_1 on the two sides no longer match, although they still agree on $\text{liveIn}(s_2, L)$.

Lemma 7.2 (`dse_bigStep_forward`). *If $\sigma_1 \Downarrow_s \sigma'_1$ and $\text{Agree}(\text{liveIn}(s, L), \sigma_1, \sigma_2)$ then there is σ'_2 with $\sigma_2 \Downarrow_{\text{optimise}(s, L)} \sigma'_2$ and $\text{Agree}(L, \sigma'_1, \sigma'_2)$.*

Lemma 7.3 (`dse_bigStep_backward`). *Suppose $\sigma_2 \Downarrow_t \sigma'_2$ with $t = \text{optimise}(s, L)$. If $\text{Agree}(\text{liveIn}(s, L), \sigma_1, \sigma_2)$ then there is σ'_1 with $\sigma_1 \Downarrow_s \sigma'_1$ and $\text{Agree}(L, \sigma'_1, \sigma'_2)$.*

Proof sketch. The forward lemma is an induction on the source derivation generalising L and σ_2 , with the same case analysis as Theorem 4.2 except that the target derivation is constructed rather than inverted. For a dead assignment the target statement is `skip`, the target state is unchanged, and the source update is to a variable outside L . The loop case uses the fixed point

$X = \text{liveIn}(\text{while } e \text{ do } s, L)$ as the invariant exactly as before, with the body hypothesis at live-out X and the loop hypothesis at L .

The backward lemma is an induction on the target derivation generalising s , L and σ_1 , with the hypothesis $t = \text{optimise}(s, L)$ carried explicitly because the target statement is not syntactically a constructor application. In each case the proof inspects s and uses the rewrite lemmas for optimise to discover which source constructors can produce the observed target constructor: a target **skip** comes either from a source **skip** or from a dead assignment, a target assignment comes only from a live assignment with the same variable and expression, and compound target statements come only from the corresponding source constructor with optimised components. The loop case feeds the hypothesis $t = \text{optimise}(\text{while } e \text{ do } s, L)$ back into the inner loop hypothesis unchanged, because the optimised loop is its own continuation. $\square$

Theorem 7.1 results from applying both lemmas with $\sigma_1 = \sigma_2$, for which agreement on any set is immediate, and translating between termination and big-step derivations.

7.2 Proof by simulation

A forward simulation associates source and target configurations through an invariant that is preserved by every source step. For dead store elimination the obvious invariant is that the target statement is the optimisation of the current source statement relative to the fixed observation set, and that the two states agree on the live-in set of the current source statement:

$$\text{Match}_L((s, \sigma_1), (t, \sigma_2)) :\Leftrightarrow t = \text{optimise}(s, L) \wedge \text{Agree}(\text{liveIn}(s, L), \sigma_1, \sigma_2).$$

This is **Match** in **Liveness/Simulation/Relation.lean**. The observation set L never changes during the computation; it is the residual statement s that changes, and the live-in set of the residual is the agreement set at that point.

Lemma 7.4 (Step simulation, `step_simulation`). *If $\text{Match}_L(c_1, c_2)$ and $c_1 \rightarrow c'_1$, then there is c'_2 with $\text{Match}_L(c'_1, c'_2)$ such that either $c_2 \rightarrow c'_2$, or $c'_2 = c_2$ and the statement of c'_1 is strictly smaller than the statement of c_1 .*

Proof sketch. Induction on the source step, generalising L and c_2 . A live assignment is matched by the same assignment on the target, with equal right-hand sides by Lemma 3.2, and the two updated states agree on $L = \text{liveIn}(\text{skip}, L)$. A dead assignment is matched by no target step: the target statement is already **skip**, and the source update is to a variable outside L ; the size of **skip** is zero and the size of an assignment is one, so the measure decreases. The rule **skip**; $s \rightarrow s$ is matched by the same rule because $\text{optimise}(\text{skip}, \cdot) = \text{skip}$. The congruence rule for sequencing applies the induction hypothesis to the first component with live-out $\text{liveIn}(s_2, L)$ and lifts the result, including the size decrease, to the sequence. Conditionals evaluate their guards equally and are matched by the same branch rule. The loop unfolding step is matched by the loop unfolding step of the target: optimising **if** e **then** $(s; \text{while } e \text{ do } s)$ **else** **skip** against L produces exactly the unfolding of **while** e **do** $\text{optimise}(s, X)$, where $X = \text{liveIn}(\text{while } e \text{ do } s, L)$, because the sequence case optimises the loop first, obtaining live-in X , and then the body against X ; and the states agree on the live-in set of the unfolded statement, $FV(e) \cup \text{liveIn}(s, X) \cup L$, because each component is contained in X by Theorem 4.1(4). $\square$

The lemma has two separable characteristics. The first is that the relation is preserved by every source step with the same L , which shows that it is a simulation and not an unrolling of the theorem. The second is that the only stuttering source steps are eliminated assignments, and each of them decreases a natural-number measure of the source statement, while other steps may increase it (loop unfolding does). This is the well-founded stuttering condition: a source execution cannot contain infinitely many consecutive stuttering steps.

Theorem 7.5 (Consequences of the simulation, `Liveness/Simulation/Preservation.lean`).
 Let $\text{Match}_L((s, \sigma_1), (t, \sigma_2))$.

1. If $\text{Terminates}(s, \sigma_1, \sigma'_1)$ then there is σ'_2 with $\text{Terminates}(t, \sigma_2, \sigma'_2)$ and $\text{Agree}(L, \sigma'_1, \sigma'_2)$ (`forward_terminates`).
2. If $\text{Diverges}(s, \sigma_1)$ then $\text{Diverges}(t, \sigma_2)$ (`forward_diverges`).
3. If $\text{Terminates}(t, \sigma_2, \sigma'_2)$ then there is σ'_1 with $\text{Terminates}(s, \sigma_1, \sigma'_1)$ and $\text{Agree}(L, \sigma'_1, \sigma'_2)$ (`backward_terminates`).

Consequently $\text{ObsEquiv}(L, s, \text{optimise}(s, L))$ (`dse_correct_sim`).

Proof sketch. Item 1 is an induction on the source execution using Lemma 7.4; at the end the source statement is `skip`, so the matched target statement is $\text{optimise}(\text{skip}, L) = \text{skip}$ and the agreement set is $\text{liveIn}(\text{skip}, L) = L$.

Item 2 needs the stuttering measure. The lemma `diverges_step_target` states that if c_1 diverges and $\text{Match}_L(c_1, c_2)$ then there is a target step $c_2 \rightarrow c'_2$ and a diverging c'_1 with $\text{Match}_L(c'_1, c'_2)$. It is proved by well-founded recursion on the size of the statement of c_1 : since c_1 diverges it can step, to some c''_1 which also diverges by determinism; if the matching target move is a step we are done, and if it is a stutter then c'_1 is smaller and matches the same c_2 , so the recursive call applies. Item 2 then follows by induction on n : for every n the target has an execution of length n , obtained by taking one target step and applying the hypothesis for $n - 1$ to the new matched pair.

Item 3 uses determinism. By Theorem 2.2(4) the source either terminates or diverges. If it diverges then by item 2 so does the target, contradicting the assumed target termination through Theorem 2.2(4) again. So the source terminates in some σ'_1 , item 1 gives a target run to some σ'_2 agreeing with σ'_1 on L , and uniqueness of terminating results identifies σ'_2 with σ'_2 . $\square$

The structure of this argument shows what a one-sided forward simulation with stuttering lacks. Items 1 and 3 together provide observational equivalence, and the additional conditions enter in item 2: divergence preservation, which is item 2 and needs the measure on stuttering steps, and determinism of the target, which is used to identify the final state of a terminating target run with that of the matched run. Without the measure, the source could diverge through an infinite number of stuttering steps while the target terminates, and without determinism the target could have one run consistent with the source and another that is not. Both properties hold in this case because the language is deterministic and because eliminated assignments decrease the residual statement, and the mechanised proof uses exactly these two facts.

7.3 Examples

The artefact includes the examples from the specification as Lean definitions with their expected transformations. For loop-free programs the transformation is proved by `decide`, which executes the optimiser inside the kernel. The fixed-point iteration is defined by well-founded recursion and does not reduce in the kernel, so for a loop the proof names the fixed point and establishes it up to set equality by `liveIn_while_equiv_of`, which combines leastness with the post-fixed point property for a candidate contained in $L \cup FV(e)$; congruence of the optimiser in its live set (`optimise_congr`) then reduces the expected transformation and the live-in set of the whole program to `decide`. The test executable checks the same cases by running the solver and the verified interpreter. The nonterminating loop `while 1 do (y := y + 1; z := 0)` observed on $\{x\}$ is optimised to `while 1 do (skip; skip)`, and for this program divergence of the source from every initial state is proved by an induction showing that no big-step derivation exists, and divergence of the target follows from `dse_preserves_divergence`. The loop-carried example

```
x := 0; y := 5; while (x < 10) do (x := x + 1; y := y + 1)
```

observed on $\{x\}$ has loop fixed point $\{x\}$: the guard reads x , the body transfer at $\{x\}$ is $\{x\}$, and y never enters. Both assignments to y are removed and both to x are kept, the transformation and the empty live-in set are proved as theorems, and the test executable confirms that source and target terminate with $x = 10$ from the zero state.

8 Mechanisation in Lean 4

The implementation is written for Lean v4.34.0 and is independent of third-party packages. It consists of eighteen modules totalling around 1800 lines, arranged as syntax and states; semantics; variable sets; observations; free variables; the analysis in four modules (domain vocabulary, fixed point, transfer, soundness); the optimisation in three modules (transformation, correctness, idempotence); the simulation in two modules; the executable interpreter and pretty printer; and three example modules. A test executable runs the example modules and checks the observable outputs of source and target under the verified interpreter.

Three choices determined the design of the mechanisation. The first is the use of lists to represent variable sets, which avoids the need for Mathlib’s `Finset`. As a result, equalities between live sets are set equalities instead of syntactic ones, which the congruence theorem `optimise_congr` covers, and in return everything in the development, including the finite-height termination argument, depends on the core library alone. The second is making the transfer function dependently typed, with $\text{liveIn}(s, L) \subseteq L \cup \text{vars}(s)$ in its type, so that the loop case can discharge the boundedness condition of the solver. The third is describing divergence as the existence of executions of every length, which makes the equivalence with non-termination constructive and gives a convenient induction principle for the simulation argument; the infinite-trace variant is proved equivalent using choice.

The trusted core is the Lean kernel combined with the three standard axioms. Applying `#print axioms` to `dse_correct`, `dse_correct_sim`, `dse_preserves_divergence`, `liveness_sound`, `lfp_fixed`, `lfp_least` and `optimise_idem` results in exactly `propext`, `Classical.choice` and `Quot.sound`; `Step.deterministic` requires no axiom at all, and the interpreter theorems rely on `propext` only. There is no `sorry`, `admit`, `native_decide` or `axiom` declaration in the repository. Beyond the kernel, nothing needs to be trusted apart from the meaning of the definitions of `Step`, `Terminates`, `Diverges`, `Agree` and `ObsEquiv`; these five are the statements a reader has to read to verify that the theorems say what this paper claims.

9 Limitations and future work

The language is deliberately restrictive. It has no procedures, arrays, pointers or input and output, so the only visible behaviours are termination itself and the final values of a fixed set of variables. All expressions are total and affect nothing else, which is why Lemma 3.2 is the only semantic fact about them that the proofs need. For a language with partial operations such as division, either the semantics would have to distinguish failure from divergence, or the analysis would have to count every expression that might fail as live even when its value is unused. The observation set is constant for the whole program rather than attached to program points, which is enough for a whole-program transformation but would need generalising before the pass could be applied to procedure bodies with calling conventions.

The simulation construction yields the same theorem as the derivation-based proof and is included because it exposes the divergence argument. It would also be the only viable route for a language with nondeterminism, where the backward direction of Theorem 7.5 could no longer rely on determinism and would need a backward simulation of its own.

The optional extension of the specification to a control flow graph with a worklist solver has not been started. The pieces that would carry over are the solver, which assumes only an arbitrary bounded monotone functional on a finite universe, and the dependency lemma. The pieces to add are a lowering from the structured language into labelled blocks together with a proof that this mapping preserves observational behaviour, liveness formulated as a set of dataflow constraints over the graph, and a proof that the worklist solution fulfils those constraints. The structured correctness theorem could then be reused only if the lowering and the transformation commute, which is itself a theorem to prove rather than a property to assume.

References

- [1] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 14–25, 2004.
- [2] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs (TYPES 2004)*, Revised Selected Papers, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81, 2006.
- [3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.
- [4] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635, 2021.
- [5] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.
- [6] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1973)*, pages 194–206, 1973.
- [7] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 42–54, 2006.
- [8] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [9] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
- [10] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.
- [11] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.