Witnessing Program Transformations

Kedar S. Namjoshi[1] and Lenore D. Zuck[2]

[1] Bell Laboratories, Alcatel-Lucent
kedar@research.bell-labs.com
[2] University of Illinois at Chicago
lenore@cs.uic.edu

Abstract. We study two closely related problems: (a) showing that a program transformation is correct and (b) propagating an invariant through a program transformation. The second problem is motivated by an application which utilizes program invariants to improve the quality of compiler optimizations. We show that both problems can be addressed by augmenting a transformation with an auxiliary *witness generation* procedure. For every application of the transformation, the witness generator constructs a relation which guarantees the correctness of that instance. We show that *stuttering simulation* is a sound and complete witness format. Completeness means that, under mild conditions, every correct transformation induces a stuttering simulation witness which is strong enough to prove that the transformation is correct. A witness is self-contained, in that its correctness is independent of the optimization procedure which generates it. Any invariant of a source program can be turned into an invariant of the target of a transformation by suitably composing it with its witness. Stuttering simulations readily compose, forming a single witness for a sequence of transformations. Witness generation is simpler than a formal proof of correctness, and it is comprehensive, unlike the heuristics used for translation validation. We define witnesses for a number of standard compiler optimizations; this exercise shows that witness generators can be implemented quite easily.

1 Introduction

An optimizing compiler is commonly structured as a sequence of passes. Each pass has a source program, which is analyzed and transformed to a target program, which then becomes the source for the next pass in the sequence. By augmenting the analysis phase of an optimization pass with information from externally supplied program invariants, it is possible to significantly enhance the quality and the effectiveness of the optimization.

To illustrate this point, consider a program which uses McCarthy's 91 function [12], which we write as $M91(x)$. The original function is doubly recursive, but has the simple property that the result is 91, if $x \leq 100$, and is $(x - 10)$ otherwise. Suppose that a programmer supplies this invariant, perhaps as part of a larger correctness proof. A compiler may then replace an invocation of this function, say `M91(a)`, with the substantially simpler conditional statement: `if (a <= 100) then 91 else (a-10)`.

F. Logozzo and M. Fähndrich (Eds.): SAS 2013, LNCS 7935, pp. 304–323, 2013.

Program invariants that enable new and improved optimization may arise from multiple sources: they may be computed by a static program analysis, be supplied as part of a correctness proof, or be generated by the analysis phase of an earlier optimization pass. The key technical challenge is to accurately propagate an invariant through multiple optimization passes. The difficulty arises because an optimization may alter program structure in arbitrary ways. For instance, a dead-code elimination removes portions of the program, expression simplification may add fresh variables and statements, and loop optimization reorders statement executions. Therefore, an invariant cannot simply be copied over unchanged from the source to the target of an optimization.

Moreover, one would like a generic and systematic propagation procedure which works for all optimizations. The questions of correctness and propagation are closely related: if there is no assurance that an optimization is correct, a target program invariant cannot be derived from an invariant for its source program, but must be computed afresh.

In this work we suggest a methodology which resolves both questions. We propose that every optimization[1] procedure is augmented with an auxiliary *witness generator*. For each instance of optimization, the generator constructs a *witness relation* between the target and source programs which guarantees correctness for that instance. We show that a *stuttering simulation* relation forms a sound and complete witness format. Stuttering simulation has several advantages. First, checking if a relation is a stuttering simulation can be done by considering only single program steps (even if stuttering is unbounded), resulting in a generic, easily implemented, and independent procedure to check for the correctness of a transformation. Second, stuttering simulation is closed under composition; thus, a sequence of witnesses, corresponding to a sequence of transformations, can be collapsed into a single witness for the entire sequence. Third, we show that a source program invariant can be propagated to the target program simply by computing its pre-image with respect to the witness relation. And, finally, we show that this format is complete: under mild conditions, a valid stuttering simulation relation can be defined for every correct transformation.

Unlike witness propagation, witness generation is not expected to be performed automatically. It assumes accesss to the optimization code and familiarity with the procedure. The additional effort required is compensated for with a better optimization that can utilize externally supplied invariants, and whose correctness can be proved independently with theorem provers.

Witness generation differs in crucial respects from the known alternatives to showing correctness of compiler optimizations. Formally proving the correctness of a transformation over all legal inputs is a daunting task[2]. Moreover, a correctness proof does not directly result in a method for propagating invariants.

[1] In this paper, we use "transformation" and "optimization" interchangeably.

[2] The remarkable effort described in [10] shows how much work is needed to construct correctness proofs for an optimizing compiler. As another estimate of the difficulty, the implementation of sparse conditional constant propagation requires over 2000 lines of C++ code in LLVM [9].

Translation validation (TV) (cf. [24]) employs heuristics to guess a witness relation for every instance of an *unknown* transformation. The heuristics, however, may fail to produce a witness for some instances.

Witness generation falls in-between these two options. Crucially, we assume full knowledge of the optimization procedure, as for formal correctness proofs, but define a generator to construct a witness for every run of the optimizer, as with TV. Full knowledge of the optimization procedure eliminates the need for heuristics, while generating a witness for each run is significantly simpler than constructing a correctness proof. The possible drawback is in the overhead of witness generation and the need to check a witness for correctness.

```
L1: y := 3;
L2: x := 10;
L3: x := 20;
L4: y := 2*x + y;
```

(a) source

```
L1: y := 3;
L3: x := 20;
L4: y := 2*x + y;
```

(b) target

Fig. 1. Dead-code elimination

The use of stuttering simulation is a departure from the common method of showing refinement, which is to establish a simulation relation from the target to the source program. Simulation is, however, incomplete: for instance, the dead-code elimination transformation shown in Figure 1 cannot be shown correct with a standard simulation relation, as the target has fewer instructions than the source. Our proof that stuttering simulation is complete is a specialization of results [1,14] on the completeness of refinement mappings; the details of this connection are laid out in Section 2. The witness relations defined in the completeness proof are necessarily complex. As we show in Section 3, however, many common optimizations may be witnessed with simple relations. This is because the complexity lies in the analysis phase which is used to determine whether a transformation is feasible, rather than in the transformation itself. A witness generator can re-use the information gathered in the analysis to define a witness.

To summarize, our contributions in this work are as follows:

- We propose augmenting each optimization pass with a *witness generator*, which creates a witness relation for every run of the optimizer.
- We show that stuttering simulation is a sound and (under mild conditions) complete witness format. As a consequence, witness checking can be made independent of the optimizations being considered.
- We show how to propagate program invariants using a stuttering simulation witness. The construction preserves inductiveness: an inductive invariant for the source turns into an inductive invariant for the target program;
- We show how to define witnesses generators for several standard compiler optimizations. The generator procedure freely uses analysis information that has been gathered for the optimization.

2 Transformations and Witnesses

We define the notion of correctness for a program transformation, and show that establishing a stuttering simulation relation from target to source is a sound and complete method for establishing correctness. We also show how to propagate invariants across a transformation using witnesses.

2.1 Background and Notation

Following Dijkstra and Scholten[8], the notation $[\varphi]$ for a formula φ represents that φ is a validity. For clarity, we often omit displaying the variables that a predicate depends on; thus, for instance, we may write $[f \ \Rightarrow \ g]$ instead of $[f(x,y) \ \Rightarrow \ g(y)]$ or the even more verbose $(\forall x, y : f(x,y) \ \Rightarrow \ g(y))$.

The *inverse* of a binary relation R is written as R^{-1}. The composition of relations R and S, written $R;S$, is the relation $\{(u,w) \mid (\exists v : (u,v) \in R \ \wedge \ (v,w) \in S)\}$. For a relation R on $D \times E$ and a predicate θ on E, the notation $\langle R \rangle \theta$ defines the set $\{d \in D \mid (\exists e : e \in E : (d,e) \in R \ \wedge \ e \in \theta)\}$. Its negation dual, denoted $[R]\theta$, defines the set $\{d \in D \mid (\forall e : e \in E \ \wedge \ (d,e) \in R : e \in \theta)\}$.

For a program A and predicate φ, $\mathbf{wlp}(A, \varphi)$ is the weakest liberal precondition operator, and $\mathbf{wp}(A, \varphi)$ is the weakest precondition operator, both defined in [7].

2.2 Programs and Transformations

Example programs in this paper are written in a C-like notation. For the formal framework, it is simpler to consider a program as a symbolic transition system.

Definition 1 (Program). *A program is described as a tuple* $(V, \Theta, \mathcal{T})$*, where*

- V *is a finite set of (typed)* state variables*, including a distinguished* program location variable*,* π*,*
- Θ *is an* initial condition *characterizing the initial states of the program,*
- $\mathcal{T}$ *is a* transition relation*, relating a state to its possible successors.*

A program *state* is a type-consistent interpretation of its variables. For a state s and a variable $v \in V$, we denote by $s[v]$ the value that s assigns to v. The transition relation is denoted syntactically as a predicate on V and V', which is a primed copy of V. For every variable x in V, its primed version x' refers to the value of x in the successor state.

There is a unique initial program location, S, such that $[\Theta \rightarrow (\pi = \mathsf{S})]$, and a unique terminal program location, F, such that $[\mathcal{T} \wedge (\pi = \mathsf{F}) \rightarrow \mathit{false}]$. An *initial state* is one where the location is S; a *final state* is one where the location is F; all other states are *intermediate* states. We assume that a program has no direct transition from an initial to a final state, and that there are no transitions to an initial state.

We assume that the transition relation of a program is *complete*; that is, for every non-final state s, there is a state s' such that $\mathcal{T}(s, s')$ holds, and that a final

state has no successor. We also assume that the transition relation is *location-deterministic*, in that there is a unique transition between any two locations. Formally, $[(\mathcal{T}(s,t) \ \wedge \ \mathcal{T}(s,v) \ \wedge \ t[\pi] = v[\pi]) \ \Rightarrow \ t = v]$. This allows non-determinism in the sense of Dijkstra's **if-fi** and **do-od** constructs where multiple guards may be true at a state, since the successor states have different locations.

A *computation* of a program is a maximal finite or infinite sequence of states $\sigma\colon s_0, s_1, \ldots$, where s_0 is an initial state and every two consecutive states on σ are related by the transition relation. Maximality implies that the last state of σ (if any) is a final state.

The notion of correct implementation ("program B implements program A") is parameterized with respect to a *compatibility* relation from the state space of B to the state space of A. Intuitively, this suggests how the initial and final states of a B-computation correspond to similar states of A.

We give some examples of compatibility relations. A renaming transformation maps every variable of program A, say x_i, to a corresponding variable, say y_i, and replaces all occurrences of the x-variables with their corresponding y-variables. The compatibility relation is simply the conjunction of terms $(x_i = y_i)$, for all i. A different transformation may replace one variable, x_0, of A with a bit-vector $b_0, \ldots, b_{31}$ in B, while renaming all other variables $x_1, x_2, \ldots$ to corresponding variables $y_1, y_2, \ldots$ as in the renaming transformation. The compatibility relation is the conjunction of $(x_0 = \sum_{k=0}^{31} b_k \cdot 2^k)$ with the terms $(x_i = y_i)$ for $i \geq 1$.

Definition 2 (Computation Matching). *Let A and B be programs, and σ^B and σ^A be maximal computations of B and A respectively. Then σ^B is* matched *by σ^A up to a compatibility relation α if the following all hold:*

- *The initial states of σ^B and σ^A are related by α,*
- *If σ^B is terminating, so is σ^A and their final states are related by α, and*
- *If σ^B is infinite then so is σ^A.*

The definition does not require that intermediate states of σ^B and σ^A are compatible. We make this simplifying choice for this work because, typically, an optimizing transformation preserves sequential semantics, which depends only on initial and final states. It is straightforward to modify this definition to require matching of intermediate states.

Definition 3 (Implementation). *Given programs A and B and a compatibility relation α, we say that B* implements *A up to α if for every maximal computation of B, there is a maximal computation in A that matches it up to α.*

Requiring non-terminating computations of B to be matched to non-terminating computations of A rules out pathological "implementations" where B does not terminate on any input.

Theorem 1. *The implementation formulation has the following properties.*

1. *(Composition) If B implements A up to α, and C implements B up to β, then C implements A up to $\beta; \alpha$.*

2. *(Preservation) If B implements A up to α, then for any predicates pre and $post$, if $[\langle\alpha^{-1}\rangle pre \Rightarrow \mathbf{wlp}(A, [\alpha^{-1}]post)]$ then $[pre \Rightarrow \mathbf{wlp}(B, post)]$, and $[\langle\alpha^{-1}\rangle pre \Rightarrow \mathbf{wp}(A, [\alpha^{-1}]post)]$ then $[pre \Rightarrow \mathbf{wp}(B, post)]$.*

Proof. (Sketch)The composition property follows directly from the definitions. We sketch a proof of the preservation property for **wlp**. Suppose there is an initial state u of B which satisfies pre and a terminating computation of B from u which ends in a final state v. As B implements A up to α, there is a terminating computation of A starting from an initial state s and ending in a final state t; these states match u and v, respectively, by α. As $(u, s) \in \alpha$, the state s satisfies $\langle\alpha^{-1}\rangle pre$. Therefore, the state t satisfies $[\alpha^{-1}]post$. As $(v, t) \in \alpha$, it follows that v satisfies $post$.

The proof for **wp** uses the identity $[\mathbf{wp}(S, q) \equiv \mathbf{wlp}(S, q) \wedge \mathbf{wp}(S, true)]$ from [7]. It remains to prove that $[pre \Rightarrow \mathbf{wp}(B, true)]$ under the assumption $[\langle\alpha^{-1}\rangle pre \Rightarrow \mathbf{wp}(A, true)]$. Consider state u and s as before. If $\mathbf{wp}(B, true)$ does not hold at u, there is an infinite computation from u in B, which must be matched by an infinite computation from s in A. As s satisfies $\langle\alpha^{-1}\rangle pre$, this leads to a contradiction. □

A *transformation* is a *partial* function on the set of programs. A transformation τ is *correct* up to a parametric compatibility function α if for every program A in its domain, $B = \tau(A)$ implements A up to $\alpha(A)$. In practical terms, a transformation is partial because it need not apply to all programs. Indeed, much of the effort in compiler optimization is on the analysis required to determine whether a particular transformation can be applied.

2.3 Stuttering Simulation

The definition of implementation requires matching computations of unbounded length, which is difficult to verify. A more directly verifiable formulation is in terms of simulation, which matches single transitions.

It is simpler to define simulation in terms of a *transition system*, given by a tuple (S, I, R), where S is a set of states, I is the subset of initial states and $R \subseteq S \times S$ is a transition relation. A program $(V, \Theta, \mathcal{T})$ induces the transition system where the states are interpretations of V, the initial states are those satisfying Θ, and the relation R is that defined symbolically by $\mathcal{T}$.

Definition 4 (Step Simulation). *Given transition systems B and A, a relation $X \subseteq S_B \times S_A$ is a* step simulation *if (a) every state in I_B is related by X to some state in I_A, and (b) for every u, s and v such that $(u, s) \in X$ and $(u, v) \in R_B$, there is some $t \in S_A$ such that $(s, t) \in R_A$ and $(v, t) \in X$.*

The following theorem is immediate.

Theorem 2 (Step Soundness). *For programs B and A and a compatibility relation α, the program B implements A up to α if there is a step simulation X from B to A, such that (1) for every initial state s^B of B, there is an initial*

state s^A of A such that (s^B, s^A) is in both X and α, and (2) for every final state t^B of B, if $(t^B, t^A) \in X$ then t^A is a final state of A and $(t^B, t^A) \in \alpha$.

Thus, checking the single-transition conditions of step simulation, together with the two additional conditions of Theorem 2, suffices to show that B is an implementation of A up to α. These checks can be encoded as validity questions and (possibly) resolved with a decision procedure.

Step simulation implies that matching finite computations have the same length. As pointed out in the introduction, this requirement makes it impossible to show the correctness of certain transformations using step simulation. Stuttering simulation [6] relaxes this condition to allow successive non-empty segments of the two computations to match by X, as is illustrated in Figure 2(a). However, these segments may be of arbitrary length, which makes it difficult to check a candidate relation.

For this reason, we use an equivalent single-step definition of stuttering simulation, formulated in [16] and refined in [14]. This requires, in addition to the state relation, a ranking function whose value decreases strictly at each stuttering step, ensuring that every maximal stuttering segment is finite. We use a simpler form of the definition, which is illustrated in Figure 2(b).

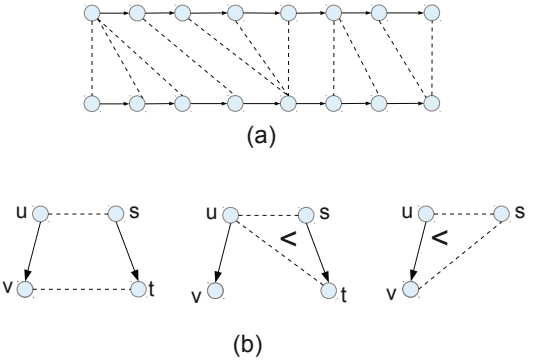

Fig. 2. Stuttering Simulation. Part (a) shows matching computations; states related by X are connected with a dashed line. Part(b) illustrates the single-step formulation.

Definition 5 (Stuttering Simulation with Ranking). *Consider transition systems B and A, a relation $X \subseteq S_B \times S_A$, a well-founded domain $(D, \prec)$, and a partial ranking function, rank : $S_B \times S_A \to D$. The relation X is a* stuttering simulation *if (a) every state in I_B is related by X to some state in I_A, and (b) for every $u, v \in S_B$ and $s \in S_A$ such that $(u, s) \in X$ and $(u, v) \in R_B$, one of the following holds:*

- *There is t such that $(s, t) \in R_A$ and $(v, t) \in X$, or*

- *There is t such that $(s,t) \in R_A$ and $(u,t) \in X$ and $rank(u,t) \prec rank(u,s)$ (stuttering in A), or*
- *$(v,s) \in X$ and $rank(v,s) \prec rank(u,s)$ (stuttering in B)*

The strict decrease in rank on every stuttering step ensures that any stuttering sequence must be of finite length.

2.4 Soundness and Completeness of Stuttering Simulations

Definition 6 (Witness). *Let A and B be programs with α as a compatibility relation from B to A. An* α-witness *for (A,B) is a relation X from the state space of B to that of A which is a stuttering simulation and satisfies the additional conditions*

- *For every initial state u of B, there is an initial state s of A such that (u,s) is in X and α, and*
- *For every final B-state v, and any A-state t, if $(v,t) \in X$, then t is final for A and $(v,t) \in \alpha$.*

It is a well known fact that stuttering simulations are closed under composition and union (see [13] for a proof). It is straightforward to show that the union and the composition of witnesses satisfies the two additional conditions. Hence, we obtain the following theorem.

Theorem 3. *[Closure Properties] The union of witness relations is a witness. If X is an α-witness for (A,B) and Y is a β-witness for (B,C), then $Y;X$ is a $\beta;\alpha$-witness for (A,C).*

Theorem 4. *[Soundness] If X is an α-witness for the program pair (A,B), then B implements A up to α.*

Proof. Suppose that σ^B is a maximal computation of B with start state u. By the first condition of Definition 6, there is an initial state s of A that is related to u by both X and α. As X is a stuttering simulation, one can inductively construct a maximal computation σ^A of A from s which matches σ^B. Formally, matching requires that that σ^A and σ^B can be partitioned into corresponding non-empty segments where any pair of states in corresponding segments are related by X. A full proof showing the inductive construction can be found in [16].

Matching implies that the first condition of Definition 2 is met by the choice of initial state for σ^A. We now show the second condition. Suppose σ^B is finite, so its last state, say v, is final for B. This state is X-related to some state, say t, on σ^A. By condition (2) of the witness definition, t is final for A, and therefore the last state of σ^A, and (v,t) is in α. This meets the second condition of Definition 2. On the other hand, if σ^B is infinite, so is σ^A, by construction. This meets the third condition of Definition 2.

Thus, every maximal computation of B has a matching computation in A, so that B implements A up to α by Definition 3. □

In [1], Abadi and Lamport showed that establishing a simulation is complete for showing language containment after the two transition systems are augmented with history and prophecy variables. Prophecy variables are needed to account for stuttering and branching. In [14], Manolios sketches a proof that stuttering simulation is complete when augmented with history and prophecy variables, where prophecy variables are used only to account for non-determinism. We prove in Theorem 5 that stuttering simulation is complete for programs with deterministic transitions, where unbounded non-determinism is allowed in the choice of initial state. The proof shows that prophecy variables are unnecessary in this situation, while history can be folded into the definition of stuttering simulation. As compiler optimizations are performed on deterministic internal representations, the assumptions made are valid in practice.

Theorem 5. *[Completeness] Consider programs B and A both of which have a deterministic transition relation. If B implements A up to α, there is an α^h-witness for the pair (A^h, B^h). Here, P^h and α^h are augmentations of program P and relation α with respect to a history variable h.*

We first sketch out the idea of the proof. By the definition of implementation, every computation σ of B has a matching computation δ of A. As A and B are deterministic, the computations are non-branching. The stuttering simulation relation connects initial states of the two computations, final states (if any), and every pair of intermediate states. A history variable is used to differentiate occurrences of the same program state on different computations; as there is no branching, it suffices to record the initial state of a computation.

Proof. (of Theorem 5) Given a program $P = (V, \Theta, \mathcal{T})$, construct P^h, an extension of P with a history variable h. The history variable is an array that records a value for every program variable. The new program has variable set $V^h = V \cup \{h\}$, transition relation $\mathcal{T}^h = \mathcal{T} \wedge (h' = h)$, and initial condition $\Theta^h = \Theta \wedge (\wedge x : x \in V : h(x) = x)$. The new initial condition ensures that the initial values of all program variables are recorded in the history variable.

For a state s of an extended program, the initial state corresponding to it is denoted $init(s)$. In this state, the location is S, every program variable x has the value stored for it in the history h, i.e., $init(s)[x] = (s[h])(x)$, and the history variable has the value stored in the history; i.e., $init(s)[h] = s[h]$. The state of the original program which state s corresponds to is called $orig(s)$. In this state, the location is the location of s, and every program variable has the value it has in s, i.e., $orig(s)[x] = s[x]$ for all $x \in V$.

Suppose that programs B and A have been extended in this manner to B^h and A^h respectively. Determinism and completeness of transitions ensures that there is a single computation from every initial state. The relation X between B^h and A^h is defined as follows. For a state u of B^h and a state v of A^h, the pair (u, v) is in X iff the following conditions hold:

- (Reachability) u is on the computation from $init_B(u)$ in B^h and v is on the computation from $init_A(v)$ in A^h,

- (Matching) The computation in B starting at $orig_B(init_B(u))$ is matched (as in Definition 2) by the computation in A starting at $orig_A(init_A(v))$.
- (Position) u and v are either both initial states, both final states, or both intermediate states.

The function $rank(u, v)$ is defined only if $(u, v) \in X$ and u and v are both on a path to a final state. It has the value (m, n) where m is the number of steps to the final state from u and n is the number of steps to the final state from v. (By determinism, at most one final state can be reached from any state.) The comparison function compares rank values point-wise.

We claim that X is a stuttering simulation relation. Consider a pair (u, v) in X. The definition of X implies that for any descendant u' of u (including $u' = u$) and descendant v' of v (including $v' = v$), the pair (u', v') satisfies the reachability and matching conditions. This follows as the definition of the history variable and its update imply that $init_B(u) = init_B(u')$ and $init_A(v) = init_A(v')$. Hence, in the following, we focus on re-establishing the Position condition for successor states.

(1) If u is on a path to a final state, so must v by the Matching constraint. Consider a transition (u, u'). By the definitions, u' must be either an intermediate or a final state.

Suppose u' is a intermediate state. If v is a intermediate state, then $(u', v) \in X$; moreover, $rank(u', v) \prec rank(u, v)$ as u' is closer to its final state than u. If v is an initial state, its unique successor v' must be an intermediate state, so $(u', v') \in X$. It is not possible for v to be final state, as u would also have to be a final state by the definition of X, and would not have a successor.

Suppose u' is a final state. Then u (and therefore v) must be an intermediate state. If v has a final successor v', then $(u', v') \in X$. If not, then v has a successor v' that is an intermediate state. Then $(u, v') \in X$; moreover, $rank(u, v') \prec rank(u, v)$ as v' is closer to its final state than v.

(2) If u has no path to a final state, neither can v by the path matching condition. Consider a transition (u, u'). If u is an initial state, so is v, by the definition of X, so there is an intermediate successor v' of v such that $(u', v') \in X$. If u is an intermediate state, so are v and u'; hence, by completeness of the transition relation, v has an intermediate successor v', and $(u', v') \in X$.

This proof establishes that X is a stuttering simulation. We now establish the two additional conditions that are required for X to be a witness. Define α^h so that $(u, v) \in \alpha^h$ iff $(orig_B(u), orig_A(v)) \in \alpha$.

Consider an initial state u of B^h. Then $s = orig_B(u)$ is an initial state of B which, as B implements A up to α, is related to an initial state t of A such that $(s, t) \in \alpha$. Consider the initial state v of A^h formed by extending t with the initial value of the history variable. Then $orig_A(v) = t$, so that $(u, v) \in \alpha^h$.

Suppose u, v are states such that $(u, v) \in X$ and u is final. From the Position condition, v must also be final. As the history variable is purely auxiliary, the computation from $init_B(u)$ to u has a corresponding computation in B from $orig_B(init_B(u))$ to $orig_B(u)$. Similarly, the computation from $init_A(v)$ to v has a corresponding computation in A from $orig_A(init_A(v))$ to $orig_A(v)$. By the

Matching condition, these computations match, so that $(orig_B(u), orig_A(v))$ is in α, as $orig_B(u)$ is a final state of B. Hence, $(u, v) \in \alpha^h$. □

2.5 Invariant Propagation

Program invariants may arise from multiple sources: for instance, they may be supplied externally via a correctness proof or a static analysis of the source program, or computed internally as part of the analysis phase of an optimization pass. Having a witness relation helps to propagate both types of invariants for use in later stages of optimization. Note that the propagated invariant does not depend on the ranking function used to show that W is a stuttering simulation.

Theorem 6. *Let W be a stuttering simulation witness for a transformation from program A to program B. If θ is an invariant for A, the set $\langle W \rangle \theta$ is an invariant for B. Moreover, if θ is inductive, so is $\langle W \rangle \theta$.*

Proof. Let σ be a computation of B. From the stuttering simulation definition, there is a computation δ of A such that every state on σ is related to some state on δ. As δ is a computation of A, every state along it satisfies θ. It follows that every state on σ satisfies $\langle W \rangle \theta$. Hence, the assertion $\langle W \rangle \theta$ is an invariant for B.

Now assume that θ is inductive; we show that $\langle W \rangle \theta$ is inductive as well. The base case, that every initial state of B satisfies $\langle W \rangle \theta$, holds as all such states are related to some initial state of A. Consider any state u of B which satisfies $\langle W \rangle \theta$. Therefore, there is a state s of A such that $(u, s) \in W$ and s satisfies θ. Consider a transition in B from u to v. By the stuttering simulation definition, v corresponds by W to a state t that is reachable by a finite (possibly empty) path in A from s. As θ is inductive, every state on this path, including t, satisfies θ; hence, v satisfies $\langle W \rangle \theta$. □

2.6 Computational Questions

Consider a sequence of transformations, with respective witnesses $W_1, W_2, \ldots, W_k$. An invariant θ for the source program may be transferred in stages to the invariant $\langle W_K \rangle (\langle W_{K-1} \rangle (\ldots \langle W_2 \rangle (\langle W_1 \rangle \theta) \ldots))$ for the target. As the pre-image operator distributes over composition, this is equivalent to $\langle W_K; \ldots; W_2; W_1 \rangle \theta$. Witnesses are closed under composition by Theorem 3, so letting $X = W_K; \ldots; W_2; W_1$ be the witness for the entire transformation sequence, this expression can be written succinctly as $\langle X \rangle \theta$.

An interesting question is whether to perform invariant propagation in an eager or lazy manner. Eager propagation transfers the invariant for each stage. Since not all stages necessarily use the transferred invariant, an alternative is to transfer an invariant only when needed.

We expect that the primary use of a transferred invariant will be to check the validity of Hoare-triples under the invariant. The checks, therefore, have the shape $[(\langle W \rangle \theta \ \wedge \ pre) \ \Rightarrow \ \mathbf{wlp}(S, post)]$. This can be written equivalently as $[(W \ \wedge \ \theta \ \wedge \ pre \ \wedge \ S) \ \Rightarrow \ post]$, which eliminates the existential quantification in $\langle W \rangle$. The quantifier-removal is important as, for many logics, efficient decision procedures are known only for their quantifier-free fragments.

3 Witnesses for Common Optimizations

In this section we define witnesses for several standard optimizations. The optimizations are chosen for their commonality and in order to illustrate features of the witness generation. We consider conditional constant propagation, dead-code elimination, control-flow graph compression, and a number of loop optimizations. For constant propagation, the witness is a step simulation; however, dead-code elimination and control-flow graph compression requires stuttering simulation, as the target code is shorter than the original. Loop optimizations, such as interchange, tiling, and reversal require more complex witnesses which maintain invariants about the loop. In each case, witness generation makes explicit the implicit invariants gathered during the analysis.

3.1 Conditional Constant Propagation

In conditional constant propagation, the analysis algorithm does not propagate constants through conditional branches which can be derived to be "dead"; i.e., those which have a guard which evaluates to false. This produces more accurate results. For instance, in the example of Figure 3, determining that the "then" branch of the conditional is dead allows y to be a constant after the conditional.

```
L1: x := 10;
L2: y := x*x;
L3: z := 2*x + 30;
L4: if(3*z < y){
L5:  y := y+1;
L6: }else{
L7:  y := y+2;
L8: }
L9: z := y+10;
L10:
```

(a) source

```
L1: x := 10;
L2: y := 100;
L3: z := 50;
L4: skip;
L7: y := 102;
L8: skip;
L9: z := 112;
L10:
```

(b) target

Fig. 3. Conditional Constant Propagation

Constant propagation determines a set of variables that are known to be constant at each location, along with their values. This set can be represented as an assertion. For instance, at L3, the assertion is $\pi = L3 \wedge x = 10 \wedge y = 100$. The set of such assertions forms an inductive invariant of the source program.

We use a symbolic representation to specify the relation between target and source programs. For a source variable x, we use $\overline{x}$ to represent the same variable in the target program. The general shape of the witness relation for constant propagation is the following. A target state t is related to a source state s iff (a) program locations of s and t correspond, (b) all variables have identical values in s and t, and (c) the inductive invariant representing constant values holds of

the source program. In our example, for simplicity, we rename locations in the target so that the correspondence is obvious (e.g., $L3$ in the source corresponds to $L3$ in the target) but such renaming is not required.

The witness relation for the example program includes the following clause. Note that the invariant for the source program has been "folded-in" to the relation through the assertions $x = 10 \ \wedge \ y = 100 \ \wedge \ z = 50$.

$$(\pi = L4) \wedge (\overline{\pi} = L4) \wedge (x = \overline{x}) \wedge (y = \overline{y}) \wedge (z = \overline{z}) \wedge x = 10 \wedge y = 100 \wedge z = 50$$

Carrying the invariants in the relation is necessary to match transitions as required for a step simulation. For instance, the unconditional transition from the target location $L4$ to $L7$ can be matched by the conditional source transition from $L4$ to $L7$ only because the values of y and z are known to be the constant values. We obtain the following theorem.

Theorem 7. *For any correct constant propagation, the defined relation is a step simulation witness which preserves all variables.*

3.2 Dead Code Elimination (DCE)

Dead code elimination is based on an analysis of "live" variables. A variable is live at a program point if there is program path starting at that point where the variable is used before it is redefined. (All variables are considered live at S and F nodes.) If the transition from location m to location n assigns a value to a variable v that is dead (i.e., not live) at n, the assignment is replaced with a *skip* statement. This is illustrated in Figure 4 which performs dead-code detection for the output of the conditional analysis.

```
L1: x := 10;
L2: y := 100;
L3: z := 50;
L4: skip;
L7: y := 102;
L8: skip;
L9: z := 112;
L10:
```

(a) source

```
L1: x := 10;
L2: skip;
L3: skip;
L4: skip;
L7: y := 102;
L8: skip;
L9: z := 112;
L10:
```

(b) target

Fig. 4. Dead code elimination

The result of the liveness analysis is a set, denoted $live(l)$, for each location l of the source program. The witness relation for DCE is the following. A target state t is related to a source state s if (a) the program locations are identical for s and t, and (b) every variable that is live at the source location has the same value

in target and source states – i.e., for every variable v such that $v \in \mathit{live}(s[\pi])$: $s[v] = t[\overline{v}]$. For the example programs, the relation includes the clause

$$(\pi = L3) \ \wedge \ (\overline{\pi} = L3) \ \wedge \ (x = \overline{x})$$

as only the variable x is live at $L3$.

Theorem 8. *For any correct dead code elimination, the defined relation is a step simulation witness which preserves all variables.*

Proof. Every initial state of the target is an initial state of the source. Consider a pair of related states (t, s). Let m be the common location in s and t. Now consider a transition from t to t'. There is a corresponding transition from s to s' where s' and t' have the same location, as the control flow of the program is unchanged. The transition from t to t' is either a *skip* that is a result of eliminating an assignment of a dead variable at l, or corresponds to an identical transition in the source. Note that the transition in the source must be based only on variables live at m. In the latter case, as s and t agree on the values of live variables, the result of the transition is identical in both source and target.

In the first case, the source transition from s must have the form $y := e$ for some variable y that is dead at the successor location m'. Consider a variable x that is live at m'. Hence, $x \neq y$, so that $s'[x] = s[x]$. By the skip transition, $t'[x] = t[x]$. Variable x must also be live at m, thus, $s[x] = t[x]$ by the witness relation, so that $s'[x] = t'[x]$, as desired. Finally, as all variables are considered live at `S` and `F` nodes, the two additional conditions in the witness definition hold for the compatibility relation which preserves all program variables. □

3.3 Control-Flow Graph Compression (CFG)

The output of the dead code elimination has several unnecessary skip statements. These may be removed using the rewrite rule which replaces `skip;S` by `S`, for any statement S. This compresses the control flow graph of the program. Other instances of compression may occur in the following situations: (1) a sequence such as `goto L1; L1:S` is replaced with `L1:S`, or (2) the sequence `S1;S2` replaces the sequence `S1;if (C) skip else skip;S2`. In each case, the target program is shorter than the source. There cannot, therefore, be a step simulation witness; it is necessary to introduce stuttering.

The general witness definition relates a target state t to a source state s if for all $v \neq \pi$, $s[v] = t[v]$ and either $s[\pi] = t[\pi]$ or $s[\pi]$ lies on a linear chain of skip statements starting from $s[\pi]$ in the source graph. For our example, the witness relation connects $L1$ in the target to $\{L1, L2, L3, L4\}$ in the source, and $L7$ to $\{L7, L8\}$, while $L9$ and $L10$ are connected to $L9$ and $L10$, respectively.

As for the ranking, note that every *skip*-sequence occurs in the same basic block. Hence, we can assign a rank to each stuttering pair that measures its distance from the end of the source *skip*-sequence, while non-stuttering pairs are given a sufficiently high rank. Thus, one possible ranking is $(L1, L1) \mapsto 3$, $(L1, L2) \mapsto 2$, $(L1, L3) \mapsto 1$, $(L1, L4) \mapsto 0$, $(L7, L7) \mapsto 3$, $(L7, L8) \mapsto 2$, $(L9, L9) \mapsto 3$, and $(L10, L10) \mapsto 3$.

```
L1: x := 10;
L2: skip;
L3: skip;
L4: skip;
L7: y := 102;
L8: skip;
L9: z := 112;
L10:
```

(a) source

```
L1: x := 10;
L7: y := 102;
L9: z := 112;
L10:
```

(b) target

Fig. 5. Control-Flow-Graph compression

Theorem 9. *For a correct control-flow graph compression, the defined relation is a stuttering simulation witness which preserves all variables.*

Proof. (Sketch)Suppose that target state t is related to source state s. Then location $s[\pi]$ is on a linear chain of skip statements from $t[\pi]$ in the source graph. This chain must be of bounded length; the distance to the end of the chain provides the rank function needed for the stuttering simulation proof. A transition from t is matched either by a transition from s, or by a stuttering skip transition from s to s', where s' and t are matched by the witness while the rank decreases along the step. The two additional conditions of the witness definition hold for the compatibility relation which preserves all program variables. □

As stuttering simulations are closed under composition, the witnesses for constant propagation, dead-code elimination, and control-flow graph compression can be composed to form a single witness for the transformation from the program in Figure 3(a) to the program in Figure 5(b).

4 Reordering Transformations

A *reordering transformation* is a program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement [2]. It preserves a dependence if it preserves the relative execution order of the source and target of that dependence, and thus preserves the meaning of the program. Reordering transformations include many loop optimizations including fusion, distribution, interchange, and tiling.

A generic loop can be described by the statement "**for** $\boldsymbol{i} \in \mathcal{I}$ **by** $\prec_{\mathcal{I}}$ **do** $\mathtt{B}(\boldsymbol{i})$" where $\boldsymbol{i}$ is the loop induction variable and $\mathcal{I}$ is the set of the values assumed by $\boldsymbol{i}$ through the different iterations of the loop. The set $\mathcal{I}$ can typically be characterized by a set of linear inequalities.

4.1 Loop Invariant Code Motion

This is a simple reordering transformation, also referred to as "hoisting" or "scalar promotion". In it, a statement (or a group of statements) in the loop

body $\mathtt{B}(\boldsymbol{i})$ that does not depend on any of the loop iterations is taken out of the loop body. See for example Fig. 6, which is a simplified version of an example from [15]. The assignments to a and c are not dependent on any statement in the loop body. Moreover, the loop body is executed at least once. These facts can be established by a static dependency analysis. Therefore, the two assignments may be moved before the loop without changing the overall semantics.

The stuttering simulation maps the first few statements of the target program $(L1, L12, L13)$ and the first iteration of the target loop into the first iteration of the source loop. This requires stuttering, as there are more instructions in the target program segment than in the source program segment. The corresponding symbolic (stuttering simulation) matching may thus include $(\pi = L5) \wedge (\overline{\pi} = L5) \wedge (i = \overline{i}) \wedge (a = \overline{a}) \wedge (b = \overline{b}) \wedge (c = \overline{c}) \wedge (i = 1) \wedge (a = 3) \wedge (b = 2) \wedge (c = 2)$. From the second iteration onwards, the two loops are linked in a step simulation as, by that stage, the values of a, b, and c are established as identical constants in both programs. This pattern, of matching up the first iterations of the loops using a stuttering simulation, while subsequent iterations are in a step simulation, applies to the general instance of loop invariant code motion. For these iterations, the matching may include $(\pi = L5) \wedge (\overline{\pi} = L5) \wedge (i = \overline{i}) \wedge (a = \overline{a}) \wedge (b = \overline{b}) \wedge (c = \overline{c}) \wedge (d = \overline{d}) \wedge (i > 1) \wedge (a = 3) \wedge (b = 2) \wedge (c = 2)$. An alternative treatment of this transformation can be found in [23].

```
L1:  b := 2;
L2:  for i=1 to 100 do{
L3:       a := b + 1;
L4:       c := 2;
L5:       d := (i mod 2) * c;}
L6:
```

(a) source

```
L1:   b := 2;
L12:  a := 3;
L13:  c := 2;
L2:   for i=1 to 100 do{
L5:        d := (i mod 2) * 2;}
L6:
```

(b) target

Fig. 6. Loop Invariant Code Motion

4.2 Loop Reordering Transformations

"Loop transformations" usually refer to a group of transformations that reorder the loop bodies themselves, rather than the statements inside the loop body, and have the generic form:

$$\textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\boldsymbol{i}) \quad \Longrightarrow \quad \textbf{for } \boldsymbol{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\boldsymbol{j})) \qquad (1)$$

In such a transformation, we may possibly change the domain of the loop indices from $\mathcal{I}$ to $\mathcal{J}$, the names of loop indices from $\boldsymbol{i}$ to $\boldsymbol{j}$, and possibly introduce an additional linear transformation in the loop's body, changing it from the source $\mathtt{B}(\boldsymbol{i})$ to the target body $\mathtt{B}(F(\boldsymbol{j}))$. An example of such a transformation is *loop reversal*, that can be described as

$$\textbf{for } i = 1 \textbf{ to } N \textbf{ do } \mathtt{B}(i) \quad \Longrightarrow \quad \textbf{for } j = N \textbf{ to } 1 \textbf{ (by } -1\textbf{) do } \mathtt{B}(j)$$

Here $\mathcal{I} = \mathcal{J} = [1..N]$, the transformation F is the identity, and the two orders are given by $i_1 \prec_{\mathcal{I}} i_2 \iff i_1 < i_2$ and $j_1 \prec_{\mathcal{J}} j_2 \iff j_1 > j_2$, respectively. Since we expect the source and target programs to execute the same instances of the loop's body (possibly in a different order), the mapping $F : \mathcal{J} \mapsto \mathcal{I}$ is a bijection from $\mathcal{J}$ to $\mathcal{I}$.

The work in [24] includes a comprehensive table of common loop transformations expressed in this form. There, "structure preserving" and "reordering" transformations are treated differently, here we claim that witnesses allow for uniform treatment of the two types of transformations. There, it is shown that the following *commutation conditions* suffice for a correct loop transformation:

1. The mapping F is a bijection from $\mathcal{J}$ onto $\mathcal{I}$.
2. For every $\boldsymbol{i}_1 \prec_{\mathcal{I}} \boldsymbol{i}_2$ such that $F^{-1}(\boldsymbol{i}_2) \prec_{\mathcal{J}} F^{-1}(\boldsymbol{i}_1)$, $\mathtt{B}(\boldsymbol{i}_1);\mathtt{B}(\boldsymbol{i}_2) \sim \mathtt{B}(\boldsymbol{i}_2);\mathtt{B}(\boldsymbol{i}_1)$.

Establishing simulation iteration by iteration may be difficult (perhaps even useless at times); the commutation conditions are sufficient to establish stuttering simulation for states before and after the loop body. Propagation of inner loop invariants, however, may be beneficial to perform further optimizations. While a general scheme for establishing such a transformation may require complex logics and reasoning, in many cases the obvious scheme — of replacing a source invariant $\varphi(\boldsymbol{i})$ by its counterpart $F^{-1}(\varphi(\boldsymbol{i}))$ — is correct. For example, consider the programs in Fig. 7 and let `AssertionA` be the assertion $\varphi_A(\boldsymbol{i})$: $sum = \sum_{k=1}^{i-1} a[i]$. Since for every $i = 1, \ldots, N$, $F^{-1}(i) = N - j + 1$, we replace $k = 1$ with $k = N - 1 + 1 = N$, $i - 1$ with $F^{-1}(i-1) = N - j$, and $a[i]$ with $a[j]$ to obtain $\varphi_B(\boldsymbol{j})$: $sum = \sum_{k=N-j}^{N} a[k]$ for `AssertionB`.

```
B0
L1:  sum := 0;
B1   {sum = 0}
L2:      for i=1 to N do{
****AssertionA
L3:       sum := sum + a[i];}
B2
L4:
```

(a) source

```
B0
L1:  sum := 0;
B1   {sum = 0}
L2:    for j=N to 1 by (-1) do{
****AssertionB
L3:       sum := sum + a[j];}
B2
L4:
```

(b) target

Fig. 7. Vector summation reversal

5 Discussion, Conclusions, and Related Work

Ensuring the correctness of program transformations – in particular, compiler optimizations – is a long-standing research problem. In [11], Leroy gives a nice technical and historical view of approaches to this question. A primary approach

is to formally prove each transformation correct, over all legal input programs. This is done, for example, in the CompCert project [10], and in [5], which derives and proves correct optimizations using denotational semantics and a relational version of Hoare's logic. However, formal verification of a full-fledged optimizing compiler, as one would verify any other large program, is often infeasible, due to its size, evolution over time, and, possibly, proprietary considerations. *Translation Validation* offers an alternative to full verification. The idea is to construct a *validating tool* which, after every run of the compiler, formally confirms that the target code produced is a correct translation of the source program. (Proof-carrying code [19] is related but certifies specific properties of programs.) A primary assumption of this approach is that the validator has limited knowledge of the transformation process. Hence, a variety of methods for translation validation arise (cf. [20,18,21,23,24,22]), each making choices between the flexibility of the program syntax and the set of possible optimizations that are handled. As details of the optimization are assumed to be unknown, each method employs heuristics to set up an inductive correctness proof for a run of the optimizer. This approach is, therefore, naturally limited in its reach by the heuristics that are used to compute a correctness proof.

More recently, [4] study *certificate translation*, which transforms a correctness proof of a source program into a correctness proof of the program's transformation, and *certificate analysis*, which transforms a proof of correctness from one formalism into another. In [3], a method for proving semantic equivalence programs based on relational Hoare logic is presented. While there are similarities to our use of stuttering simulation relations as witnesses, the general thrust is closer to translation validation rather than witness generation, and has similar limitations.

Our approach, while close to translation validation, differs crucially in that it supposes that the optimization procedure is known and can be examined and augmented. Hence, we suppose that the optimization procedure can be augmented with a witness generator which produces witnesses which are checked – as in the translation validation – at run-time. As the optimization process is visible to the witness generator, the generator is able to make use of auxiliary invariants derived by the optimizer in order to produce a witness. This implies that witness generation is, in principle, applicable to any optimization. The particular form of witness that is considered here ensures that it is complete; hence, a witness checker may be written once and reused for the witnesses produced by a variety of transformations. The completeness result applies to deterministic programs. This may seem like a limitation; however, program optimizations are, for the most part, applied to deterministic sections of code, although the full program may have non-determinism from inputs and thread-level scheduling.

In practice, limits may arise from the complexity of the witness relation that must be produced. For instance, the logics needed to express the witness may not have decision procedures, so that fully automated witness checking is not possible. However, in several cases – a selection of which is presented in Section 3 – witnesses can be expressed in terms of simple logics which are solvable using

current SMT solvers. An interesting question, which we plan to address in future work, is the extent to which specialized forms of witnesses may be generated for efficient checking.

We also state and provide a solution for the problem of invariant propagation. This is prompted by recent (ongoing) work to crowd-sourced formal verification, which will enable an application to use manually generated invariants to enhance and extend compiler optimizations. However, one need not rely solely on crowd-sourcing or expert intervention for invariants; sound static analysis tools often produce deep invariants for program code, especially loops, which are not uncovered by the quick analysis carried out inside a compiler.

Invariant propagation is a special case of the proof propagation that is discussed in [17]; however, that work considers only propagation of inductive invariants through a step simulation. Theorem 6 extends the propagation result to general invariants and stuttering simulation. While invariant propagation of a kind is standard in optimizing compilers (e.g., the results of a points-to analysis on the source program may be used in several subsequent optimizations), to the best of our knowledge, the problem of invariant propagation had not been addressed in the general form discussed here. An interesting practical issue with invariant propagation is whether it should be performed in an eager or lazy manner, as discussed briefly in Section 2.5.

In this paper, we have considered a simple, procedure-free model of programs. A large number of standard optimizations fit this model. Extending witness generation and checking to inter-procedural optimizations is a topic of ongoing work. In current work, we are developing witness generators for several of the commonly applied optimization routines in LLVM [9], using SMT solvers to check the correctness of the generated witnesses.

Acknowledgements. This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. 82(2), 253–284 (1991)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann (2002)
3. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013)
4. Barthe, G., Kunz, C.: An abstract model of certificate translation. ACM Trans. Program. Lang. Syst. 33(4), 13 (2011)

5. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL, pp. 14–25 (2004)
6. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. 81(1), 13–31 (1989)
7. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. CACM 18(8) (1975)
8. Dijkstra, E., Scholten, C.: Predicate Calculus and Program Semantics. Springer (1990)
9. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88 (2004), Webpage at `llvm.org`
10. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL, pp. 42–54. ACM (2006)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
12. Manna, Z., McCarthy, J.: Properties of programs and partial function logic. Journal of Machine Intelligence 5 (1970)
13. Manolios, P.: Mechanical Verification of Reactive Systems. PhD thesis, University of Texas at Austin (2001)
14. Manolios, P.: A compositional theory of refinement for branching time. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 304–318. Springer, Heidelberg (2003)
15. Muchnick, S.: Advanced Compiler Design & Implementation. Morgan Kaufmann, San Francisco (1997)
16. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: Ramesh, S., Sivakumar, G. (eds.) FST TCS 1997. LNCS, vol. 1346, pp. 284–296. Springer, Heidelberg (1997)
17. Namjoshi, K.S.: Lifting temporal proofs through abstractions. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 174–188. Springer, Heidelberg (2002)
18. Necula, G.: Translation validation of an optimizing compiler. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation, PLDI 2000, pp. 83–95 (2000)
19. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: OSDI, pp. 229–243. ACM (1996)
20. Pnueli, A., Siegel, M., Shtrichman, O.: The code validation tool (CVT)- automatic verification of a compilation process. Software Tools for Technology Transfer 2(2), 192–201 (1998)
21. Rinard, M., Marinov, D.: Credible compilation with pointers. In: Proceedings of the Run-Time Result Verification Workshop (July 2000)
22. Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: PLDI, pp. 295–305 (2011)
23. Zuck, L.D., Pnueli, A., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. J. UCS 9(3), 223–247 (2003)
24. Zuck, L.D., Pnueli, A., Goldberg, B., Barrett, C.W., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Formal Methods in System Design 27(3), 335–360 (2005)