A Framework for Formal Verification of Compiler Optimizations

William Mansky, Elsa Gunter

Compiler verification

- program correctness relies on compiler
- real compilers have bugs, and they're hard to find
- compiler optimizations are complicated and not usually verified
- goal: transformed program is *semantically equivalent*

Framework Overview

- write optimization in TRANS
  - rewrite language on CFGs
  - side conditions in CTL on CFGs
- prove correctness using Isabelle, CTL, given lemmas for TRANS
- in compiler, model-check condition before rewriting

Control Flow Graphs (for L_0)

0: x := 1

1: if x > 0 goto 4

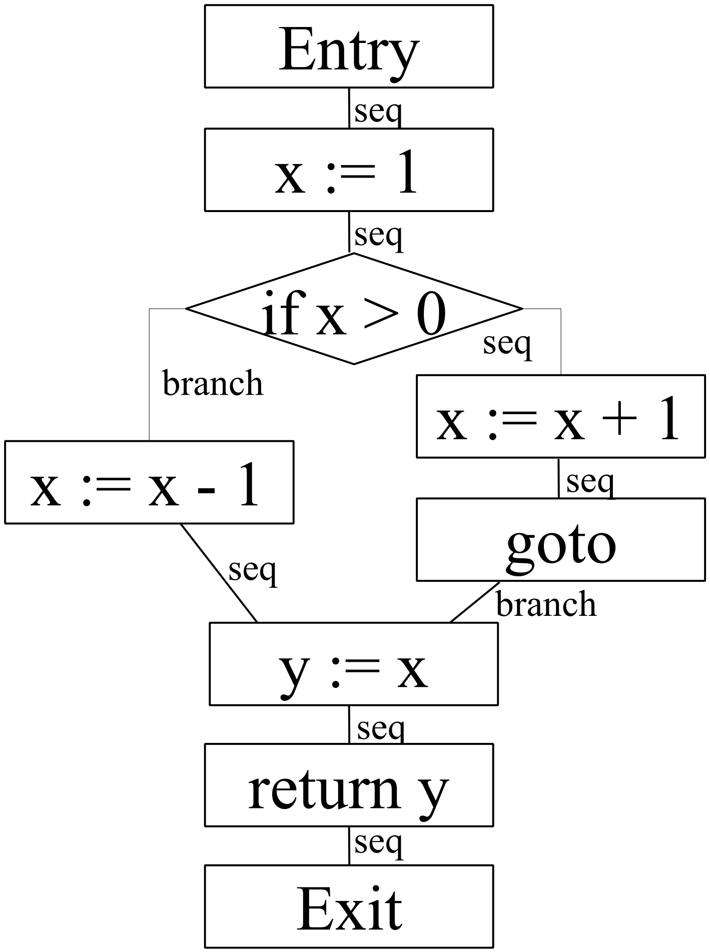
2: x := x + 1

3: goto 5

4: x := x - 1

5: y := x

6: return y



The TRANS Language

replace x := e with skip (transformation)

if

$\neg EX(E(\neg def(x)\ U\ (use(x) \wedge \neg node(n))))$ (CTL)

@ n (node)

i.e., if there is no path forward along which x is used before it is redefined

- first presented by Kalvala et al.
- we gave full formal semantics in Isabelle

TRANS actions (on CFGs)

- add_edge(n,m,e) – add an edge from n to m labeled e
- remove_edge(n,m,e) – remove an edge from n to m labeled e
- replace n with $p_1,\ldots,p_k$ – replace the instr at n with instrs $p_1,\ldots,p_k$
- split_edge(n,m,e,q) – insert q in the middle of the edge from n to m
- These actions may not preserve CFGs

TRANS Strategies

- analogous to LCF tacticals
- match φ in T, T_1 □ T_2, T_1 then T_2, apply_all T
- amended recursive semantics for apply_all:
  - inductively define apply_some(f, τ, G) to apply f some number of times
  - [apply_all T](τ, G) = apply_some([T], τ, G) \

$$\{G' \mid G'' \in [T](\tau, G') \wedge G'' \neq G'\}$$

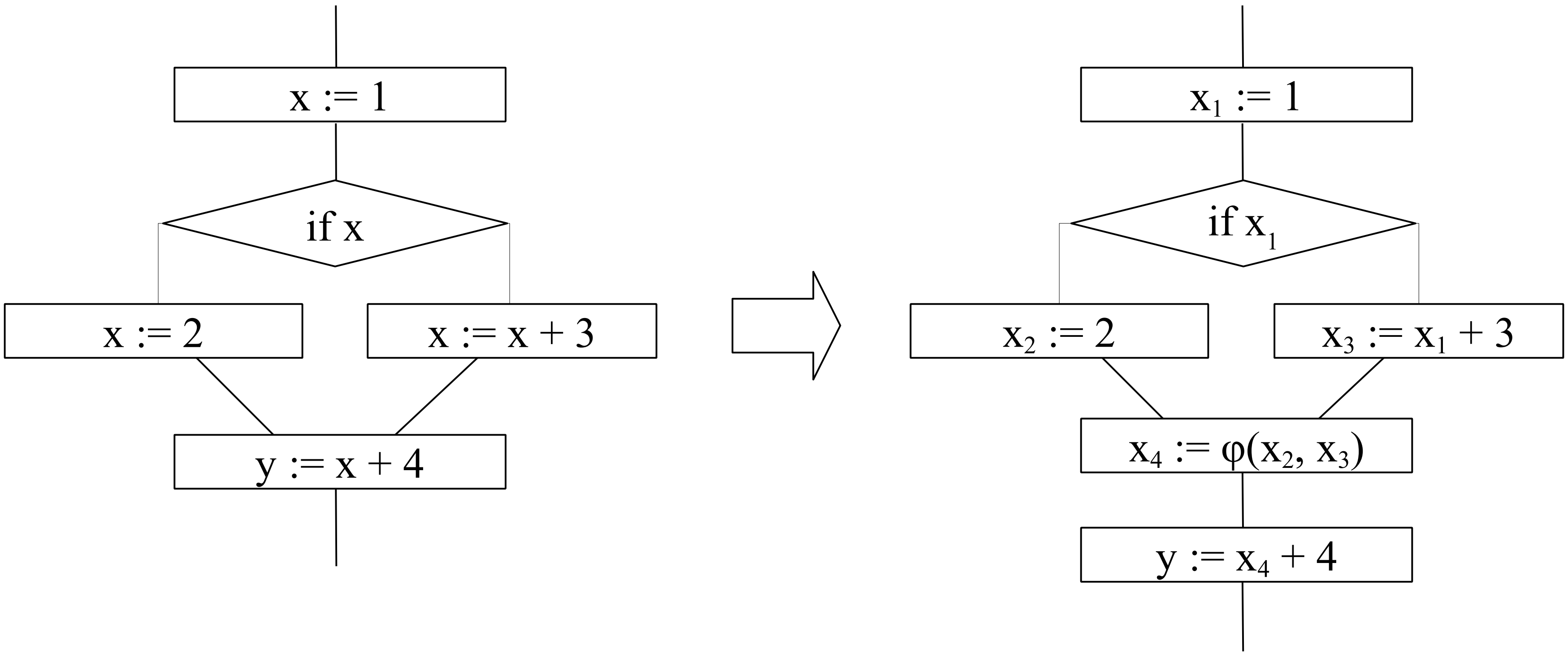
Correctness of a transformation

- A TRANS formula on a graph G defines a set S of transformed graphs
- We can define language semantics as transition system on CFGs
- A transformation is *semantics-preserving* if for all $G' \in S$, $G \to^* v$ iff $G' \to^* v$

Case Study: SSA

- Kalvala et al. have already expressed simple optimizations
- Static Single Assignment (SSA) is a common transformation in optimizing compilers
- extends language of source programs
- no known verified algorithm

Static Single Assignment



Using the Framework

- step 1: write SSA conversion in TRANS
  - side conditions capture basic logic of SSA
  - try to maximize modularity
- step 2: verify conversion
  - first priority is semantic preservation
  - also want to show result is SSA
  - must first show preservation of CFGs

Correctness of SSA

- step 0: define parameter language
  - $L_1 = L_0 + \varphi$-functions
- SSA in TRANS over L_1 has four steps
  - add_index – change each $x := e$ to $x_i := e$
  - add_phi – add φ-functions at join points
  - update – change each use of x to the x_i that reaches it
  - refactor – change the x_i's to new variables

Sample proof step: add_phi

Theorem: If G is a CFG with no φ-functions, each application of add_phi preserves the semantics of G

Proof: by induction on program trace (stuttering bisimulation)

Precondition: each var instance has a unique definition (1) + graph has no φ-functions

Postcondition: (1) + φ-nodes are the only nodes reached by multiple definitions of the same var + graph has no non-empty φ-functions

Results

- the first verified TRANS optimization
- the first verified SSA conversion
- revised and formalized TRANS semantics
- proved various general lemmas about CFG preservation, reaching defs, etc.
- modular proof with lightweight pre- and post-conditions
- hope to extend to parallel optimizations

Related Work

- Kalvala et al. (2009): defined TRANS, expressed opts.
- Visser et al. (1999): rewrite-based opts., no conditions or verification
- Leroy (2006, 2009): opts. based on dataflow analysis, limited changes to structure
- Blech & Glesner (2004): verified code generation from SSA

apply_some

- solution: inductively define apply_some(f, τ, G)
- G $\in$ apply_some(f, τ, G)

$$\frac{G' \in f(\tau, G) \qquad G'' \in \text{apply\_some}(f, \tau, G')}{G'' \in \text{apply\_some}(f, \tau, G)}$$

Step 1: add_index

apply_all (replace n with (x_k := e)

if

varlit(x) & stmt(x := e) @ n & freshNew(x, k))

Step 2: add_phi

apply_all (replace n with x_k := φ(), i

if

stmt(i) @ n & multi_defs(x) @ n & freshNew(x, k) & ~(n1 is n2) & (EXR node(n1) & EXR node(n2)) @ n & A(stmt(y := φ(s)) & ~(x is y) U ~stmt(y := φ(s)) @ n))

Step 3: update

apply_all (match reaches(x_k) @ n in

replace n with i[x_k] if stmt(i[x]) @ n □

replace n with $x_{k'}$:= φ(x_k, s) if

stmt($x_{k'}$:= φ(s)) @ n & ~(x_k in s))

- reaches is defined in terms of until

Step 4: refactor

apply_all (match fresh z in

(replace n with z := e if

stmt(x_k := e) @ n □

replace n with z := φ(s) if

stmt(x_k := φ(s)) @ n) then

replace n with i[z] if stmt(i[x_k]) @ n)

add_index

```
lemma add_index_ok:
assumes more_nodes and "CFG G"
shows "preserves_results add_index G"
```

Proof: by induction on program trace (step-by-step correspondence)

Precondition: graph contains no φ-functions

Postcondition: each var instance has a unique definition

add_index

Theorem: If G is a CFG, each application of add_index preserves the semantics of G

Proof: by induction on program trace (step-by-step correspondence)

Precondition: graph contains no φ-functions

Postcondition: graph contains no φ-functions + each var instance has a unique definition (1)

add_phi

Theorem: If G is a CFG with no φ-functions, each application of add_phi preserves the semantics of G

Proof: by induction on program trace (stuttering bisimulation)

Precondition: (1)

Postcondition: (1) + graph has no non-empty φ-functions + φ-nodes are the only nodes reached by multiple definitions of the same var

update

Theorem: If G is a CFG in which (1) holds and φ-nodes are the only nodes reached by multiple definitions of the same var, FULL application of update preserves the semantics of G

Proof: by induction on program trace (one-to-one correspondence)

Precondition: none

Postcondition: (1) + each var use is indexed with its reaching definition (2) + each φ-function holds all reaching instances of its base var (3)

refactor

Theorem: If G is a CFG in which (1) and (2) hold and, in any execution trace of G, the reaching instance at each φ-function is in the body of the φ-function, each application of refactor preserves the semantics of G

Proof: by induction on program trace (bisimulation based on refactored memories)

Precondition: (3)

Postcondition: (1) + all indexed vars replaced

SSA conversion

```
theorem conversion_ok: "[| more_nodes;
recoverable G; to_SSA_graph G0 = G |]
==> preserves_results conversion G"
```

Proof: by combination of correctness properties for each step

Postcondition: (1) + all indexed vars have been replaced, implying that resulting graph is SSA