Towards Verification of GVN-CSE on a Functional Intermediate Language with System Calls

Sigurd Schneider, Sebastian Hack, Gert Smolka

Work done during a visit to Xavier Leroy at Inria Paris-Rocquencourt

Talk at Inria Paris-Rocquencourt

02.06.2014
Overview

- imperative
- functional
- machine
Overview

- imperative
- functional + syscalls
- machine
Overview

- imperative
- functional + syscalls
- machine
Overview

- imperative
- functional + syscalls
- machine

SSA-Construction

SSA-Destruction
Overview

value optimizations

functional

+ syscalls
Overview
In this talk

1. Program equivalence
   - System calls
   - Inductive proofs
     * Allow modification of function signatures

2. Semantic specification of a value optimization
   - suitable for GVN-CSE
   - inductive correctness proof

Syntax of IL

Adding system calls

\[ s, t ::= \text{let } x = e \text{ in } s \]

\[ | \text{if } e \text{ then } s \text{ else } t \]

\[ | x \]

\[ | \text{fun } f \overline{x} = s \text{ in } t \]

\[ | f \overline{e} \]

\[ | \text{let } x = \text{extern } f \overline{e} \text{ in } s \]

- First-order functional language with tail-call restriction
- Expressions in conditionals an function applications
- Non-deterministically choose return value for extern functions
Semantics of IL/I and IL/F

Common Part

\[\begin{array}{c}
\text{Op} \\
V \vdash e \Downarrow v \\
\rightarrow \quad L \mid V \mid \text{let } x = e \text{ in } s \\
\rightarrow \quad L \mid V^x_v \mid s
\end{array}\]

\[\begin{array}{c}
\text{If} \\
\text{val2bool}(v) = i \\
V \vdash e \Downarrow v \\
\rightarrow \quad L \mid V \mid \text{if } x \text{ then } s_0 \text{ else } s_1 \\
\rightarrow \quad L \mid V \mid s_i
\end{array}\]
Semantics of IL/I and IL/F

Difference: Application Rule of IL/F

IL/F: Lexical scoping

```
1 let x = 7 in
2 fun f () = x in
3 let x = 5 in f ()
```

IL/F-Let

```
L | V | fun f x = s in t
→ L, f := (V, x, s) | V | t
```

IL/F-App

```
L, f := (V', x, s), L' | V | f y
→ L, f := (V', x, s) | V'_{\overline{y}} | s
```

V': Closure Environment

V: Primary Environment
Semantics of IL/I and IL/F

Difference: Application Rule of IL/I

IL/I: Dynamic binding

1 \( x := 7; \)
2 \( \textbf{fun} \ f \ () = x \ \textbf{in} \)
3 \( x := 5; \ f () \)

IL/I-Let

\[
\begin{array}{c}
L \mid V \mid \text{fun } f \ x = s \ \text{in } t \\
\rightarrow \ L, \ f := (\overline{x}, s) \mid V \mid t
\end{array}
\]

IL/I-App

\[
\begin{array}{c}
L, \ f := (\overline{x}, s), \ L' \mid V \mid f \ y \\
\rightarrow \ \downarrow \ L, \ f := (\overline{x}, s) \mid V \mid \overline{V} \ y \mid s
\end{array}
\]

\( V' \): Closure Environment

\( V \): Primary Environment
Semantics of IL/I and IL/F

External calls

\[ \begin{align*}
& \text{Extern} \quad V \vdash \overline{e} \downarrow \overline{v} \quad w \in \mathbb{V} \\
& \quad L \ | \ V \ | \ let \ x = \ extern \ f \ \overline{e} \ in \ s \\
& (f,\overline{v},w) \quad \rightarrow \quad L \ | \ V_x^w \ | \ s
\end{align*} \]

- Introduces non-determinism (any \( w \in \mathbb{V} \) will do)
- *external* non-determinism: If observation know, deterministic
- *internal* non-determinism is not allowed
  - And our definitions exploit that
Overview

1. Program Equivalence
   - Bisimilarity
   - Similarity

value optimizations

functional
+ syscalls
Program Equivalence

Bisimilarity and Similarity

\[
\begin{array}{c}
\text{BS} \quad \sigma_1 \rightarrow^+ \sigma_1' \\
\sigma_2 \rightarrow^+ \sigma_2' \\
\sigma_1' \sim \sigma_2'
\end{array}
\] 

\[
\sigma_1 \sim \sigma_2
\]

\[
\begin{array}{c}
\text{BC} \quad \nu \in R \cup \{\bot\} \\
\sigma_1 \Downarrow \nu \\
\sigma_2 \Downarrow \nu
\end{array}
\]

\[
\sigma_1 \sim \sigma_2
\]

\[
\begin{array}{c}
\sigma_1, \sigma_2 \text{ activated} \\
\sigma_1 \rightarrow^+ \sigma_1' \\
\forall \sigma_1'', \sigma_1' \xrightarrow{o} \sigma_1'' \Rightarrow \exists \sigma_2'', \sigma_2' \xrightarrow{o} \sigma_2'' \land \sigma_1'' \sim \sigma_2''
\end{array}
\]

\[
\begin{array}{c}
\sigma_2 \rightarrow^+ \sigma_2' \\
\forall \sigma_2'', \sigma_2' \xrightarrow{o} \sigma_2'' \Rightarrow \exists \sigma_1'', \sigma_1' \xrightarrow{o} \sigma_1'' \land \sigma_1'' \sim \sigma_2''
\end{array}
\]

\[
\sigma_1 \sim \sigma_2
\]

\[
\begin{array}{c}
\sigma \text{ activated if there is a next step, and it is not silent}
\end{array}
\]
Program Equivalence
Bisimilarity and Similarity

\[
\begin{array}{c}
\text{BE} \\
\sigma_1 \downarrow \perp \rightarrow \\
\sigma_1 \sim \sigma_2
\end{array}
\]

- Bisimilarity (BS, BC, BN) preserves error
- Similarity (BS, BC, BN, BE) is an implementation relation
  - If first program errs, no further restriction on second
  - External events up to error must be preserved
Overview

1. Program Equivalence
   - Bisimilarity
   - Similarity

2. Inductive Proof Technique

value optimizations

functional
+ syscalls
Inductive Proof Technique

What programs do we want to prove similar?

1. \texttt{let x = 5 + 9 in}
2. \texttt{fun f (y) = y in}
3. \texttt{f (x)}

1. \texttt{fun f () = 14 in}
2. \texttt{f ()}

- Constant folding + parameter elimination
- Suppose: proof by induction on the program
- Induction hypothesis must be strong enough to show that \( f(x) \) (left) is equivalent to \( f() \) (right)
Inductive Proof Technique
Extension Lemma Enables Inductive Proofs

- \( A \) relates arguments, \( P \) relates parameters each indexed by analysis information \( a \)
- \( f := (\overline{x}, E, s), L \approx_{a,A} f' := (\overline{x}', E', s'), L' \) if
  - parameters are related: \( P a \overline{x} \overline{x}' \)
  - for \( A \) \( a \)-related arguments, \( f, f' \) equivalent (similar)
  - inductively, \( L \approx_A L' \)
- \( \overline{x}, E, s \sim_{a,A} \overline{x}', E', s' \) if \( P a \overline{x} \overline{x}' \) and for all \( \overline{y} \overline{y}' \) \( L L' \)

\[
\begin{align*}
& A a \overline{y} \overline{y}' \Rightarrow L \approx_{a,A} L' \Rightarrow (L, E[\overline{x} \mapsto \overline{y}], s) \sim (L', E'[\overline{x}' \mapsto \overline{y'}], s')
\end{align*}
\]

Lemma (Extension)

If \( L \approx_A L' \) and \( \overline{x}, E, s \sim_{a,A} \overline{x}', E', s' \) then

\[
\begin{align*}
f := (\overline{x}, E, s), L & \approx_{a,A} f' := (\overline{x}', E', s'), L' \end{align*}
\]
Suppose equivalence proof by induction on the syntax:

\[ L \approx_A L' \Rightarrow (L, E, s) \sim (L', E', s') \]
Inductive Proof Technique

Extension Lemma

1. Suppose equivalence proof by induction on the syntax:
   \( L \equiv_A L' \Rightarrow (L, E, s) \sim (L', E', s') \)

2. Critical case is function definition:
   \((L, E, \text{fun } f \bar{x} = s \text{ in } t) \sim (L', E', \text{fun } f \bar{x}' = s' \text{ in } t')\)
Inductive Proof Technique

Extension Lemma

1. Suppose equivalence proof by induction on the syntax:
   \[ L \approx_A L' \Rightarrow (L, E, s) \sim (L', E', s') \]

2. Critical case is function definition:
   \[(L, E, \text{fun } f \overline{x} = s \text{ in } t) \sim (L', E', \text{fun } f \overline{x'} = s' \text{ in } t') \]

3. \sim \text{ closed under expansion, reduce one step (left and right):}
   \[(f := (\overline{x}, E, s); L, E, t) \sim (f := (\overline{x'}, E', s'); L', E', t') \]
Inductive Proof Technique

Extension Lemma

1. Suppose equivalence proof by induction on the syntax:
   \[ L \approx_A L' \Rightarrow (L, E, s) \sim (L', E', s') \]

2. Critical case is function definition:
   \[(L, E, \text{fun } f \overline{x} = s \text{ in } t) \sim (L', E', \text{fun } f \overline{x'} = s' \text{ in } t')\]

3. \sim \text{ closed under expansion, reduce one step (left and right):}
   \[(f := (\overline{x}, E, s); L, E, t) \sim (f := (\overline{x'}, E', s'); L', E', t')\]

4. Apply IH for \(t\), but now have to show
   \[ f := (\overline{x}, E, s), L \approx_{a,A} f' := (\overline{x'}, E', s'), L' \]
Inductive Proof Technique

Extension Lemma

1. Suppose equivalence proof by induction on the syntax:
\[ L \approx_A L' \Rightarrow (L, E, s) \sim (L', E', s') \]

2. Critical case is function definition:
\[ (L, E, \text{fun } f \overline{x} = s \text{ in } t) \sim (L', E', \text{fun } f \overline{x'} = s' \text{ in } t') \]

3. \sim \text{ closed under expansion, reduce one step (left and right)}:
\[ (f := (\overline{x}, E, s); L, E, t) \sim (f := (\overline{x'}, E', s'); L', E', t') \]

4. Apply IH for \( t \), but now have to show
\[ f := (\overline{x}, E, s), L \approx_{a,A} f' := (\overline{x'}, E', s'), L' \]

5. Apply extension lemma, reduces goal to: For all \( y y' L_1 L'_1 \)
\[ \forall a y y' \Rightarrow L_1 \approx_{a,A} L'_1 \Rightarrow (L_1, E[\overline{x} \leftarrow \overline{y}], s) \sim (L'_1, E'[\overline{x'} \leftarrow \overline{y'}], s') \]
Inductive Proof Technique

Extension Lemma

1. Suppose equivalence proof by induction on the syntax:
   \[ L \approx_A L' \Rightarrow (L, E, s) \sim (L', E', s') \]

2. Critical case is function definition:
   \[ (L, E, \text{fun } f \overline{x} = s \text{ in } t) \sim (L', E', \text{fun } f \overline{x'} = s' \text{ in } t') \]

3. \( \sim \) closed under expansion, reduce one step (left and right):
   \[ (f := (\overline{x}, E, s); L, E, t) \sim (f := (\overline{x'}, E', s'); L', E', t') \]

4. Apply IH for \( t \), but now have to show
   \[ f := (\overline{x}, E, s), L \approx_{a,A} f' := (\overline{x'}, E', s'), L' \]

5. Apply extension lemma, reduces goal to: For all \( y \overline{y} L_1 L'_1 \)
   \[ \forall a \overline{y} y' \Rightarrow L_1 \approx_{a,A} L'_1 \Rightarrow (L_1, E[\overline{x} \mapsto \overline{y}], s) \sim (L'_1, E'[\overline{x'} \mapsto \overline{y'}], s') \]

6. Discharge by IH for \( s \)
Inductive Proof Technique
Extension Lemma

- Extension lemma works for
  - both bisimilarity and similarity
  - IL/F (shown before)
  - IL/I (analogous, but slightly different definition)

- Examples proven with extension lemma
  - Dead variable elimination
  - Value Optimizations (up next)

- Optimizations that remove code are no problem
Overview

1. Program Equivalence
   - Bisimilarity
   - Similarity

2. Inductive Proof Technique

3. Value Optimizations
Value Optimizations
Considerations for a correctness criterion for value optimizations

- When can an expression \( e \) be replaced by an expression \( e' \)?

  *If \( e' \) evaluates in at least the environments occurring during program execution to the same value as \( e \)*

- What if \( e \) does not evaluate to a value?

  *Impose no requirement on \( e' \)*

- How to characterize the environments that occur during program execution?

  *Restrict to environments that satisfy a certain set of equations*
Value Optimizations

Some semantic definitions

- Define $E \models e = e'$ to hold if $e$ and $e'$ evaluate to the same value or are both undefined under $E$
- Lift to sets of equations $\Gamma$ in point-wise manner: $E \models \Gamma$
- Define $E \models e \sqsubseteq e'$ to hold if whenever $e$ evaluates to a value under $E$, then $e'$ evaluates to the same value
- Define approximation $\Gamma \models e \sqsubseteq e'$ as

$$\forall E, \ E \models \Gamma \implies E \models e \sqsubseteq e'$$
Value Optimization
A word on decidability

- $E \models \Gamma$ decidable (for finite $\Gamma$, simple enough expression language, and finite value domain)
  - Challenge: efficient decision procedure
  - In this talk: decidability not required

- Semantic specification is suitable for a wide range of value optimizations:
  - GVN-based common subexpression elimination (not done)
  - copy propagation (done)
  - conditional constant propagation (done)
Soundness for Value Optimizations

Judgement

\[ \Lambda \mid \Delta \vdash \mathit{vopt} \ s / s' : \Gamma \]

- \( \Lambda \): information about functions
- \( \Delta \): set of defined variables
- \( s \): source program
- \( s' \): translated program
- \( \Gamma \): set of equations

Under assumptions \( \Lambda \) about the functions appearing in \( s \) and \( s' \), and assuming all variables in \( \Delta \) are defined, \( s' \) simulates \( s \) in every environment that satisfies the equations \( \Gamma \).

- inductively defined
- requires \( s \) to be renamed apart
Soundness for Value Optimizations

Rules

\[ \Lambda \mid \Delta \vdash \text{vopt } s / s' : \Gamma \]

\[ \frac{\Gamma \models e \sqsubseteq e' \quad \mathcal{V}(e') \subseteq \Delta}{\Lambda \mid \Delta \cup \{x\} \vdash \text{vopt } s / s' : \{x = e, x = e'\} \cup \Gamma} \]

\[ \frac{\Lambda \mid \Delta \vdash \text{vopt } \text{let } x = e \text{ in } s / \text{let } x = e' \text{ in } s' : \Gamma}{\Lambda \mid \Delta \vdash \text{vopt } \text{let } x = e \text{ in } s / \text{let } x = e' \text{ in } s' : \Gamma} \]
Soundness for Value Optimizations

Rules

\[ \Lambda, \Delta \vdash vopt\ s\ /\ s' : \Gamma \]

\[ \Gamma \models e \subseteq e' \quad \forall (e') \subseteq \Delta \]

\[ \begin{array}{c}
\text{Op} \quad \frac{\Lambda, \Delta \cup \{x\} \vdash \text{vopt } s\ /\ s' : \{x = e, x = e'\} \cup \Gamma}{\Lambda, \Delta \vdash \text{vopt let } x = e \text{ in } s\ /\ \text{let } x = e' \text{ in } s' : \Gamma}
\end{array} \]

- \( \Gamma \) may become inconsistent
  - No \( E \) with \( E\ y = 0 \) will satisfy \( \{x = y \cdot y/y, x = y\} \)
  - But if \( E\ y = 0 \) behavior undefined after executing \( y \cdot y/y \)
  - Strengthens what \( \Gamma \models e \subseteq e' \) means here
Soundness for Value Optimizations

Rules

\[ \begin{align*}
\Lambda, f : (\overline{x}, \Delta, \Gamma_f, \Gamma') & \mid \Delta \vdash \text{vopt } t / t' : \Gamma \\
\Lambda, f : (\overline{x}, \Delta, \Gamma_f, \Gamma') & \mid \Delta \cup \overline{x} \vdash \text{vopt } s / s' : \Gamma' \cup \Gamma_f \\
\Lambda & \mid \Delta \vdash \text{vopt } \text{fun } f \overline{x} = s \text{ in } t / \text{fun } f \overline{x} = s' \text{ in } t' : \Gamma
\end{align*} \]

1 \text{ fun } f (y) = y \text{ in } \\
2 \text{ let } x = \ldots \text{ in } \\
3 f (x)

- \( \mathcal{V}(\Gamma_f) \subseteq \Delta \cup \overline{x} \) required for scoping
- \( \mathcal{V}(\Gamma') \subseteq \Delta \) because \( \Gamma \) could be inconsistent
Soundness for Value Optimizations

Rules

\[
\text{App} \quad \frac{\Gamma \models \overline{y} \subseteq \overline{y'} \quad \Gamma \Rightarrow \Gamma_f[\overline{x} \rightarrow \overline{y'}]}{\Lambda, \ f : (\overline{x}, \Delta_f, \Gamma_f, \Gamma'_f), \ \Lambda' \ | \ \Delta \vdash \text{vopt} \ f \overline{y} / f \overline{y'} : \Gamma}
\]

- \( \Gamma \models \overline{y} \subseteq \overline{y'} \) means either both lists fully defined and equivalent, or first list contains at least one expression that is undefined.
- \( \Gamma_f[\overline{x} \rightarrow \overline{y'}] \) does not mention parameters anymore.
Soundness for Value Optimizations

Rules

\[
\frac{
\Gamma \not\models e \neq 0 \Rightarrow \Lambda \mid \Delta \vdash \text{vopt } s / s' : \Gamma \cup \{e \neq 0\}
}{
\Gamma \not\models e = 0 \Rightarrow \Lambda \mid \Delta \vdash \text{vopt } t / t' : \Gamma \cup \{e = 0\} \quad \Gamma \vdash e \sqsubseteq e'
}
\]

\[
\frac{
\text{Cond} \quad \Lambda \mid \Delta \vdash \text{vopt if } e \text{ then } s \text{ else } t / \text{if } e' \text{ then } s' \text{ else } t' : \Gamma
}{
\text{Supports } \textit{conditional} \text{ analyses}
}
\]

- If according to \( \Gamma \), a branch is unreachable, no proof obligation
- Exploit knowledge about value of condition inside branch
**Value Optimizations**

Correctness proofs for value optimization

- **Proof without vopt**
  1. Specify sound analysis result (s.t. efficiently decidable)
  2. Write transformation
  3. Prove correctness semantically

- **Proof with vopt**
  1. Specify sound analysis result (s.t. efficiently decidable)
  2. Write transformation
  3. Prove that vopt holds given sound analysis result

- **Examples proven with vopt**
  - Copy Propagation
  - Sparse Conditional Constant Propagation (in this talk)
  - GVN-CSE (not finished yet)
Overview

1. Program Equivalence
   - Bisimilarity
   - Similarity

2. Inductive Proof Technique

3. Value Optimizations

4. Constant Propagation
POPL 1991: Sparse Conditional Constant Propagation (SCCP)

Sparse
- Analysis tracks one global mapping $\mathcal{V} \rightarrow \mathcal{V} \cup \{\top\}$
- Sparseness is one of the original SSA-promises (and we achieve it)

Conditional
- Use analysis information to identify unreachable branches in conditionals during fixpoint computation
- Accommodated by Cond-rule

More powerful than applying dead code elimination and constant propagation individually
Constant Propagation with Conditional Branches

MARK N. WEGMAN and F. KENNETH ZADECK
IBM T. J. Watson Research Center

Conditional

1 fun f(x) =
2   if x then x
3   else f(x - 1)
4 in f(5)

↓ Optimization

1 fun f() = 5
2 in f()
Constant Propagation with Conditional Branches

MARK N. WEGMAN and F. KENNETH ZADECK
IBM T. J. Watson Research Center

Conditional

```haskell
fun f (x) =
  if x then x
  else f (x - 1)
in f (5)
```

↓ Optimization

```haskell
fun f () = 5
in f ()
```

Uses conditions

```haskell
if x = 5 then x
else y
↓ Optimization
if x = 5 then 5
else y
```
Sparse Conditional Constant Propagation
How do we do it?

- **vopt** does not support dead code elimination
- Two phases: optimization + dead variable elimination (DVE)
  - Analysis is conditional
  - Optimization leaves the conditional intact, but replaces the condition with constant
  - Run DVE afterwards (removes conditionals with constant condition)
  - Same principle for dead variables
- DVE is proven with extension lemma by induction (material for another talk)
Sparse Conditional Constant Propagation

Example

```plaintext
let x = 5 + 9 in
fun f (y) = y in
f (x)
```

↓ SCCP

```plaintext
let x = 14 in
fun f (y) = 14 in
f (14)
```

↓ DVE

```plaintext
fun f () = 14 in
f ()
```
Soundness for SCCP

Judgement

\[ \Lambda \vdash \text{sccp} \; s : \kappa \]

- \( \Lambda \) information about functions
- \( \kappa \) constants \( \mathcal{V} \rightarrow \mathcal{V} \cup \{\top\} \)
- \( s \) source program

**Under assumptions** \( \Lambda \) **about the functions appearing in** \( s \)

**variables in** \( s \) **can be replaced according to** \( \kappa \)

in every environment that refines \( \kappa \) on \( \mathcal{V}(s) \)

- inductively defined
- requires \( s \) to be renamed apart
Soundness for Value Optimizations

Rules

\[
\begin{align*}
\text{Cond} & \quad \text{val2bool}(\llbracket e \rrbracket \kappa) \neq 0 \Rightarrow \Lambda \vdash \text{sccp} \ s : \ \text{upd} \ \kappa \ e \ 1 \\
\text{val2bool}(\llbracket e \rrbracket \kappa) \neq 1 \Rightarrow \Lambda \vdash \text{sccp} \ t : \ \text{upd} \ \kappa \ e \ 0 \\
\Lambda \vdash \text{sccp} \ \text{if} \ e \ \text{then} \ s \ \text{else} \ t : \ \kappa
\end{align*}
\]

\[
\text{ upd } \kappa \ e \ b = \begin{cases} 
\kappa[x \to 0] & b = 0 \land e \equiv x \\
\kappa[x \to c] & b = 1 \land e \equiv x = c \\
\kappa[x \to c] & b = 0 \land e \equiv x \neq c
\end{cases}
\]
Soundness for Value Optimizations

Rules

\[\begin{align*}
\text{val2bool}(\llbracket e \rrbracket_\kappa) \neq 0 & \Rightarrow \Lambda \vdash \text{sccp } s : \text{upd } \kappa e 1 \\
\text{val2bool}(\llbracket e \rrbracket_\kappa) \neq 1 & \Rightarrow \Lambda \vdash \text{sccp } t : \text{upd } \kappa e 0 \\
\Lambda \vdash \text{sccp if } e \text{ then } s \text{ else } t : \kappa
\end{align*}\]

- $\kappa$ can be turned into a set of equations $\llbracket \kappa \rrbracket_\Delta$ (restricted to variables from $\Delta$)
- $\llbracket \kappa \rrbracket_\Delta \not\equiv e = 0 \Rightarrow \text{val2bool}(\llbracket e \rrbracket_\kappa) \neq 0$
- $\llbracket \kappa \rrbracket_\Delta \cup \{e \neq 0\} \Rightarrow \llbracket \text{upd } \kappa e 1 \rrbracket_\Delta$
- Similar obligations for alternative branch
Conclusion

1. Inductive method scales to interesting transformations
   - Dead variable elimination
   - Value optimizations

2. Constant Propagation (SCCP) in a functional setting
   - Sparse analysis as promised by SSA
   - Inductive Proof

3. Challenges
   - GVN fixpoint analysis
   - Performance
Barthe, Gilles, Delphine Demange, and David Pichardie (2012). “A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert”. In: *Programming Languages and Systems - 21st European Symposium on Programming*. Ed. by Helmut Seidl. LNCS. Tallinn, Estonia.


Abstract

We present a semantic specification of a common subexpression elimination (CSE) based on global value numbering (GVN) in the setting of a functional intermediate language. The language includes system calls and expressions that may err (division by zero). Semantic preservation for GVN-CSE is shown via induction. So far, we have implemented and verified copy propagation and constant folding against the specification; the implementation of GVN analysis is in progress.