Simple Relational Correctness Proofs for Static Analyses and Program Transformations

Nick Benton

Enrico Steffinlongo

Università Ca' Foscari - Computer science

May 29, 2015

Proving correctness of Program optimization

Lot of work on functional languages program optimization especially in

- formalization
- validation

Few work on imperative programming languages

- seems trivial
- but i's not

This work proposes three systems (one type based, and two Hoare-logics based) to prove correctness of optimization transformations.

Optimization transformations

What is the optimization of a program?
Transformation of a program to a semantically equivalent one in order to reduce the time used to compute, or to decrease the resources used.
Typical imperative program optimization includes:

- constant propagation
- dead-code elimination
- program slicing
- loop unrolling

Example

`X := 7;`
`Y := Y + 1;`
`X := 7;`
`Z := X;`
	$\Rightarrow$	`X := 7;`
		`Y := Y + 1;`
		`X := 7;`
		`Z := 7;`
			$\Rightarrow$	`X := 7;`
				`Y := Y + 1;`
				
				`Z := 7;`

Optimization transformations: examples

`X := 3`
`if X = 3 then`
`    X := 7;`
`else`
`    skip;`
`Z := X + 1;`
	`==>`	`X := 7;`
		`Z := 8;`

`if X = 3 then`
`    Y := X;`
`else`
`    Y := 3;`
	`==>`	`Y := 3`

`X := -Y`
`Z := Z - X`
`X := -X`
	`==>`	`X := Y`
		`Z := Z + X`

Table : Transformation examples

The while-language: syntax

In this work we will use the while-language.

$$
\begin{array}{rcll}
X \in \mathbb{V} & = & \{X, Y, \dots\} & \text{variables} \\
n & \in & \mathbb{Z} & \text{numbers} \\
b & \in & \mathbb{B} & \text{boolean literal} \\
iop & \in & \{+, -, \times, \dots\} \subseteq \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} & \text{integer operations} \\
bop & \in & \{<, =, \dots\} \subseteq \mathbb{Z} \times \mathbb{Z} \to \mathbb{B} & \text{integer to boolean operations} \\
lop & \in & \{\wedge, \vee, \dots\} \subseteq \mathbb{B} \times \mathbb{B} \to \mathbb{B} & \text{logical operations} \\
E & := & n | X | E \ iop \ E & \text{integer expressions} \\
B & := & b | E \ bop \ E | \texttt{not} \ B | B \ lop \ B & \text{boolean expressions} \\
C & := & \texttt{skip} | X := E | C; C & \\
 & & | \texttt{if} \ B \ \texttt{then} \ C \ \texttt{else} C & \\
 & & | \texttt{while} \ B \ \texttt{do} \ C & \text{commands} \\
S & \in & \mathbb{S} = \mathbb{V} \to \mathbb{Z} & \text{A valid State}
\end{array}
$$

- Denotational semantics of Integer expression (similar to the one for Boolean expression)

$$
\begin{array}{rcl}
[\![E]\!] & \in & \mathbb{S} \to [\![int]\!] = \mathbb{S} \to \mathbb{Z} \\
[\![n]\!]S & = & n \\
[\![X]\!]S & = & S(X) \\
[\![E_1 \; iop \; E_2]\!]S & = & ([\![E_1]\!]S) iop ([\![E_2]\!]S) \\
[\![n]\!]S & = & n
\end{array}
$$

- Denotational semantics of commands

$$
\begin{array}{rcl}
[\![C]\!] & \in & \mathbb{S} \to \mathbb{S}_\perp \\
[\![\texttt{skip}]\!] & = & \lambda S.\lceil S \rceil \\
[\![X := E]\!] & = & \lambda S.\lceil S[X \mapsto [\![E]\!]S] \rceil \\
[\![C_1; C_2]\!] & = & [\![C_2]\!]^* \circ [\![C_1]\!] \\
[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else} C_2]\!] & = & \lambda S.[\![B]\!]S \Rightarrow [\![C_1]\!] \mid [\![C_2]\!] \\
[\![\texttt{while } B \texttt{ do } C]\!] & = & fix\ f.\lambda S.[\![B]\!]S \Rightarrow f^*([\![C]\!]S) | \lceil S \rceil
\end{array}
$$

Dependency, Dead Code and Constant (DDCC)

The DDCC type system is used to prove correctness of some program transformations.

- non-standard type system
- derives typed equality between expressions and commands
- works on pairs of programs
- simple types for expressions
- maps from variables to simple types for states
- can be seen as a non-interference type system
- captures only decisions based on known variables. So it is not able to capture patterns like in example 2 in table $\{1\}$
- not capture code-motion transformation

Using DDCC we can prove the equivalence of the example 1 in table $\{1\}$

A simple type $\phi_\tau := \mathbb{F}_\tau \mid \{c\}_\tau \mid \Delta_\tau \mid \mathbb{T}_\tau$ where $\tau \in \{\texttt{int}, \texttt{bool}\}$ and c is a constant.

- $\mathbb{F}_\tau$ is an empty type
- $\{c\}_\tau$ is the type of a constant c ($5 \in \{5, 5\}_{int}$)
- Δ_τ is the type of an unknown expression (if we do not know the value of X $(X + X) \in \Delta_{int}$)
- $\mathbb{T}_\tau$ is the type of an expression that we do not care.

A state type $\Phi := - \mid \Phi, X : \phi_{int}$ is a map from variable to simple types.
Judgements are of the form

- $\vdash E \sim E' : \Phi \Rightarrow \phi_\tau$ for expressions,
- $\vdash C \sim C' : \Phi \Rightarrow \Phi'$ for commands.

We will use $\vdash C : \Phi \Rightarrow \Phi$ as shorthand for $\vdash C \sim C : \Phi \Rightarrow \Phi$

DDCC core judgements

Some simple DDCC core judgements for are:

- $\vdash n \sim n : \Phi \Rightarrow \{n\}_{int}$
- $\vdash X \sim X : \Phi, X : \phi_{int} \Rightarrow \phi_{int}$
- $\vdash \texttt{skip} \sim \texttt{skip} : \Phi \Rightarrow \Phi$
- $\dfrac{\vdash C_1 \sim C_1' : \Phi \Rightarrow \Phi' \qquad \vdash C_2 \sim C_2' : \Phi' \Rightarrow \Phi''}{\vdash (C_1; C_2) \sim (C_1'; C_2') : \Phi \Rightarrow \Phi''}$
- $\dfrac{\vdash B \sim B' : \Phi \Rightarrow \Delta_{\texttt{bool}} \qquad \vdash C \sim C' : \Phi \Rightarrow \Phi}{\vdash (\texttt{while } B \texttt{ do } C) \sim (\texttt{while } B' \texttt{ do } C') : \Phi \Rightarrow \Phi}$

These rules are able to prove relations between a phrase to itself.

DDCC judgements

Judgements used to prove equivalences of transformed programs.

sequential unit laws

$$\frac{\vdash C \sim C : \Phi \Rightarrow \Phi}{\vdash (\texttt{skip}; C) \sim C : \Phi \Rightarrow \Phi'}$$

associativity

$$\frac{\vdash (C_1; C_2); C_3 : \Phi \Rightarrow \Phi'}{\vdash ((C_1; C_2); C_3) \sim (C_1; (C_2; C_3)) : \Phi \Rightarrow \Phi'}$$

commuting conversion for conditionals

loop unrolling

self-assignment elimination

$$\vdash (X := X) \sim \texttt{skip} : \Phi, X : \phi_{int} \Rightarrow \Phi, X : \phi_{int}$$

dead-assignment elimination

$$\vdash (X := E) \sim \texttt{skip} : \Phi, X : \phi_{int} \Rightarrow \Phi, X : \mathbb{T}_{int}$$

equivalent branches for conditionals

$$\frac{\vdash C_1 \sim C_2 : \Phi \Rightarrow \Phi'}{\texttt{if } B \texttt{ then } C_1 \texttt{ else} C_2 \sim C_1 : \Phi \Rightarrow \Phi'}$$

constant folding

$$\frac{\vdash F_\tau : \Phi \Rightarrow \{c\}_\tau}{\vdash F_\tau \sim c : \Phi \Rightarrow \{c\}_\tau}$$

known branch for conditional

$$\frac{\vdash B : \Phi \Rightarrow \{true\} \qquad \vdash C_1 \sim C' : \Phi \Rightarrow \Phi'}{\texttt{if } B \texttt{ then } C_1 \texttt{ else} C_2 \sim C' : \Phi \Rightarrow \Phi'}$$

dead while

$$\frac{\vdash B : \Phi \Rightarrow \{false\}}{\vdash (\texttt{while } B \texttt{ do } C \sim \texttt{skip} : \Phi \Rightarrow \Phi}$$

divergence for while

Relational Hoare Logic

To increase the capabilities of the analysis this work proposes a system based on Relational Hoare Logics. It is based on

$$
\begin{array}{rcll}
GE & := & n \mid X\langle 1\rangle \mid X\langle 2\rangle \mid GE\ iop\ GE : & \text{generalized expressions} \\
\Phi & := & b \mid GE\ bop\ GE \mid \mathtt{not}\Phi \mid \Phi\ lop\ \Phi : & \text{relational assertions}
\end{array}
$$

Judgements are of the form:

$$\vdash C \sim C' : \Phi \Rightarrow \Phi'$$

Relational Hoare Logic: semantics

The semantics of GE, and Φ are:

$$
\begin{array}{rcl}
[\![GE]\!] & \in & \mathbb{S} \times \mathbb{S} \to \mathbb{Z} \\
[\![n]\!](S_1, S_2) & = & n \\
[\![X\langle 1\rangle]\!](S_1, S_2) & = & S_1(X) \\
[\![X\langle 2\rangle]\!](S_1, S_2) & = & S_2(X) \\
[\![E \ iop \ F]\!](S_1, S_2) & = & ([\![E]\!](S_1, S_2)) \ iop \ ([\![F]\!](S_1, S_2))
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Phi]\!] & \in & \mathbb{S} \times \mathbb{S} \\
 & = & \{(S, S') | \chi_\Phi(S, S') = true\} \\
\chi_{true}(S, S') & = & true \\
\chi_{false}(S, S') & = & false \\
\chi_{E \ bop \ F}(S, S') & = & ([\![E]\!](S_1, S_2)) \ bop \ ([\![F]\!](S_1, S_2)) \\
\chi_{\Phi \ lop \ \Phi'}(S, S') & = & (\chi_\Phi(S_1, S_2)) \ lop \ (\chi_{\Phi'}(S_1, S_2)) \\
\chi_{\mathtt{not}\Phi}(S, S') & = & \neg(\chi_\Phi(S_1, S_2))
\end{array}
$$

RHL: core

Some simple RHL core judgements are:

- $\vdash \texttt{skip} \sim \texttt{skip} : \Phi \Rightarrow \Phi$
- $\vdash X := E \sim Y := E' : \Phi[E\langle 1\rangle / X\langle 1\rangle, E'\langle 2\rangle / Y\langle 2\rangle] \Rightarrow \Phi$
- $$\frac{\vdash C_1 \sim C_1' : \Phi \Rightarrow \Phi' \qquad \vdash C_2 \sim C_2' : \Phi' \Rightarrow \Phi''}{\vdash (C_1; C_2) \sim (C_1'; C_2') : \Phi \Rightarrow \Phi''}$$
- $$\frac{\vdash C \sim C' : \Phi \wedge (B\langle 1\rangle \wedge B'\langle 2\rangle) \Rightarrow \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle)}{\begin{array}{c}\vdash (\texttt{while}\ B\ \texttt{do}\ C) \sim (\texttt{while}\ B'\ \texttt{do}\ C') : \\ \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle) \Rightarrow \Phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle)\end{array}}$$

RHL: judgements

Judgements used to prove equivalences of transformed programs.

falsity	$C \sim C' : \mathit{false} \Rightarrow \Phi$
dead-assignment elimination	$\vdash (X := E) \sim \texttt{skip} : \Phi[E\langle 1\rangle / X\langle 1\rangle] \Rightarrow \Phi$
common branch	$\dfrac{\vdash C \sim D : \Phi \wedge B\langle 1\rangle \Rightarrow \Phi' \qquad \vdash C \sim D : \Phi \wedge \texttt{not} B\langle 1\rangle \Rightarrow \Phi'}{\texttt{if } B \texttt{ then } C \texttt{ else} C' \sim D : \Phi \Rightarrow \Phi'}$
dead while	$\vdash (\texttt{while } B \texttt{ do } C \sim \texttt{skip} : \Phi \wedge \texttt{not} B\langle 1\rangle \Rightarrow \Phi \wedge \texttt{not} B\langle 1\rangle$

Questions?

Thanks!