Simple Relational Correctness Proofs for Static Analyses and Program Transformations

By Nick Benton

Enrico Steffinlongo

Università Ca' Foscari - Computer science

May 29, 2015

Soundness of Program optimization

Lot of work on functional languages especially in

- formalization
- validation

Few work on imperative programming languages

- seems trivial
- ... but i's not

Optimization transformations

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

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

```
X := 7;                 X := 7;                 X := 7;
Y := Y + 1;     ⇒       Y := Y + 1;     ⇒       Y := Y + 1;
X := 7;                 X := 7;
Z := X;                 Z := 7;                 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

Dependency, Dead Code and Constant (DDCC)

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

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 ($5 \in \{5\}_{int}$)
- 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 any more.

A state type $\Phi := - \mid \Phi, X : \phi_{int}$

while-Programs: syntax

$\mathbb{V}$	$=$	$\{X, Y, \dots\}$	a set of variables
n	$\in$	$\mathbb{Z}$	a number,
b	$\in$	$\mathbb{B}$	a boolean literal
iop	$\in$	$\{+, -, \times, \dots\} \subseteq \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	an integer operation
bop	$\in$	$\{<, =, \dots\} \subseteq \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$	an integer to boolean operation
lop	$\in$	$\{\wedge, \vee, \dots\} \subseteq \mathbb{B} \times \mathbb{B} \to \mathbb{B}$	a logical operation
E	$:=$	$n \mid X \mid E\ iop\ E$	integer expressions
B	$:=$	$b \mid E\ bop\ E \mid \texttt{not}\ B \mid B\ lop\ B$	boolean expressions
C	$:=$	$\texttt{skip} \mid X := E \mid C; C$	
		$\mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C$	
		$\mid \texttt{while}\ B\ \texttt{do}\ C$	commands
S	$\in$	$\mathbb{S} = \mathbb{V} \to \mathbb{Z}$	A valid State

while-Programs: semantics

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

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

- Denotational semantics of commands

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