Operationally-based Program Equivalence Proofs using LCTRSs

Ștefan Ciobâcă Dorel Lucanu Andrei Sebastian Buruiană
stefan.ciobaca@info.uaic.ro
dlucanu@info.uaic.ro
sebastian.buruiana@yahoo.com

Alexandru Ioan Cuza University

WPTE 2021

Outline

Functional Equivalence

Given two programs P and Q:

- do they compute the same output for the same input?

Functional Equivalence

Given two programs P and Q:

- do they compute the same output for the same input?

Many applications:

1. Compilation;
2. Optimizations;
3. Refinements.

Functional Equivalence

Given two programs P and Q:

- do they compute the same output for the same input?

Many applications:

1. Compilation;
2. Optimizations;
3. Refinements.

Many approaches, usually based on Relational Hoare Logic.

Functional Equivalence

Given two programs P and Q:
- do they compute the same output for the same input?

Many applications:

1. Compilation;
2. Optimizations;
3. Refinements.

Many approaches, usually based on Relational Hoare Logic.
Fewer results in the area of *operationally-based equivalence*:
- Given P (in a language L_1) and Q (in a language L_2), is P equivalent to Q?

Outline

Logically Constrained Term Rewriting Systems

LCTRSs have rewrite rules of the form

$$l \longrightarrow r \text{ if } \phi,$$

where l, r are terms; ϕ is a logical constraint.
terms can contain *interpreted* symbols.

Logically Constrained Term Rewriting Systems

LCTRSs have rewrite rules of the form

$$l \longrightarrow r \text{ if } \phi,$$

where l, r are terms; ϕ is a logical constraint.
terms can contain *interpreted* symbols.

Example

$$\begin{aligned}
&\mathsf{init}(\mathsf{n}) \longrightarrow \mathsf{loop}(\mathsf{n}, 2) \text{ if } \top,\\
&\mathsf{loop}(\mathsf{i} \times \mathsf{k}, \mathsf{i}) \longrightarrow \mathsf{comp} \text{ if } \mathsf{k} > 1,\\
&\mathsf{loop}(\mathsf{n}, \mathsf{i}) \longrightarrow \mathsf{loop}(\mathsf{n}, \mathsf{i} + 1) \text{ if } \neg(\exists \mathsf{k}.\mathsf{k} > 1 \wedge \mathsf{n} = \mathsf{i} \times \mathsf{k}).
\end{aligned}$$

Logically Constrained Term Rewriting Systems

LCTRSs have rewrite rules of the form

$$l \longrightarrow r \text{ if } \phi,$$

where l, r are terms; ϕ is a logical constraint.
terms can contain *interpreted* symbols.

Example

$$\begin{aligned}
&\mathsf{init}(\mathsf{n}) \longrightarrow \mathsf{loop}(\mathsf{n}, 2) \text{ if } \top,\\
&\mathsf{loop}(\mathsf{i} \times \mathsf{k}, \mathsf{i}) \longrightarrow \mathsf{comp} \text{ if } \mathsf{k} > 1,\\
&\mathsf{loop}(\mathsf{n}, \mathsf{i}) \longrightarrow \mathsf{loop}(\mathsf{n}, \mathsf{i} + 1) \text{ if } \neg(\exists \mathsf{k}.\mathsf{k} > 1 \wedge \mathsf{n} = \mathsf{i} \times \mathsf{k}).
\end{aligned}$$

LCTRSs are parametric in a theory (such as LIA, RA, BV, Theory of Arrays, etc.).

Syntax of Programming Language as Order-Sorted Signatures

$$
\begin{array}{lll}
\mathsf{Exp} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Id} \mid \mathsf{Exp}\ \texttt{binop}\ \mathsf{Exp} \mid \texttt{unop}\ \mathsf{Exp} \mid \mathsf{call}\ \mathsf{FunCall} \mid \mathsf{skip} \mid \\
 & & \mathsf{Exp};\mathsf{Exp} \mid \mathsf{Id} := \mathsf{Exp} \mid \mathsf{while}\ \mathsf{Exp}\ \mathsf{do}\ \mathsf{Exp} \mid \mathsf{if}\ \mathsf{Exp}\ \mathsf{then}\ \mathsf{Exp}\ \mathsf{else}\ \mathsf{Exp} \\
\mathsf{FunCall} & ::= & \mathsf{Id} \mid \mathsf{FunCall}(\mathsf{Exp}) \\
\mathsf{FunBody} & ::= & \mathsf{Exp} \mid \lambda \mathsf{Id}.\mathsf{FunBody} \\
\mathsf{Stack} & ::= & [] \mid \mathsf{Exp} \rightsquigarrow \mathsf{Stack} \\
\mathsf{Cfg} & ::= & \langle \mathsf{Stack}, \mathsf{Env}, \mathsf{Funcs} \rangle \\
\mathsf{Env} & ::= & \mathsf{Array}\{\mathsf{Int}\}\{\mathsf{Int}\} \\
\mathsf{Funcs} & ::= & \mathsf{Array}\{\mathsf{Id}\}\{\mathsf{FunBody}\}
\end{array}
$$

Syntax of Programming Language as Order-Sorted Signatures

$$
\begin{array}{lcl}
\mathsf{Exp} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Id} \mid \mathsf{Exp}\ \texttt{binop}\ \mathsf{Exp} \mid \texttt{unop}\ \mathsf{Exp} \mid \mathsf{call}\ \mathsf{FunCall} \mid \mathsf{skip} \mid \\
 & & \mathsf{Exp};\mathsf{Exp} \mid \mathsf{Id}{:=}\mathsf{Exp} \mid \mathsf{while}\ \mathsf{Exp}\ \mathsf{do}\ \mathsf{Exp} \mid \mathsf{if}\ \mathsf{Exp}\ \mathsf{then}\ \mathsf{Exp}\ \mathsf{else}\ \mathsf{Exp} \\
\mathsf{FunCall} & ::= & \mathsf{Id} \mid \mathsf{FunCall}(\mathsf{Exp}) \\
\mathsf{FunBody} & ::= & \mathsf{Exp} \mid \lambda\mathsf{Id}.\mathsf{FunBody} \\
\mathsf{Stack} & ::= & [] \mid \mathsf{Exp} \rightsquigarrow \mathsf{Stack} \\
\mathsf{Cfg} & ::= & \langle \mathsf{Stack}, \mathsf{Env}, \mathsf{Funcs} \rangle \\
\mathsf{Env} & ::= & \mathsf{Array}\{\mathsf{Int}\}\{\mathsf{Int}\} \\
\mathsf{Funcs} & ::= & \mathsf{Array}\{\mathsf{Id}\}\{\mathsf{FunBody}\}
\end{array}
$$

- Sorts are the nonterminals.

Syntax of Programming Language as Order-Sorted Signatures

$$
\begin{array}{lll}
\mathsf{Exp} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Id} \mid \mathsf{Exp}\ \texttt{binop}\ \mathsf{Exp} \mid \texttt{unop}\ \mathsf{Exp} \mid \mathsf{call}\ \mathsf{FunCall} \mid \mathsf{skip} \mid \\
 & & \mathsf{Exp};\mathsf{Exp} \mid \mathsf{Id} := \mathsf{Exp} \mid \mathsf{while}\ \mathsf{Exp}\ \mathsf{do}\ \mathsf{Exp} \mid \mathsf{if}\ \mathsf{Exp}\ \mathsf{then}\ \mathsf{Exp}\ \mathsf{else}\ \mathsf{Exp} \\
\mathsf{FunCall} & ::= & \mathsf{Id} \mid \mathsf{FunCall}(\mathsf{Exp}) \\
\mathsf{FunBody} & ::= & \mathsf{Exp} \mid \lambda\mathsf{Id}.\mathsf{FunBody} \\
\mathsf{Stack} & ::= & [] \mid \mathsf{Exp} \rightsquigarrow \mathsf{Stack} \\
\mathsf{Cfg} & ::= & \langle \mathsf{Stack}, \mathsf{Env}, \mathsf{Funcs} \rangle \\
\mathsf{Env} & ::= & \mathsf{Array}\{\mathsf{Int}\}\{\mathsf{Int}\} \\
\mathsf{Funcs} & ::= & \mathsf{Array}\{\mathsf{Id}\}\{\mathsf{FunBody}\}
\end{array}
$$

- Sorts are the nonterminals.
- Language constructs are operations.

Syntax of Programming Language as Order-Sorted Signatures

$$
\begin{array}{lll}
\mathsf{Exp} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Id} \mid \mathsf{Exp}\ \texttt{binop}\ \mathsf{Exp} \mid \texttt{unop}\ \mathsf{Exp} \mid \mathsf{call}\ \mathsf{FunCall} \mid \mathsf{skip} \mid \\
 & & \mathsf{Exp};\mathsf{Exp} \mid \mathsf{Id} := \mathsf{Exp} \mid \mathsf{while}\ \mathsf{Exp}\ \mathsf{do}\ \mathsf{Exp} \mid \mathsf{if}\ \mathsf{Exp}\ \mathsf{then}\ \mathsf{Exp}\ \mathsf{else}\ \mathsf{Exp} \\
\mathsf{FunCall} & ::= & \mathsf{Id} \mid \mathsf{FunCall}(\mathsf{Exp}) \\
\mathsf{FunBody} & ::= & \mathsf{Exp} \mid \lambda\mathsf{Id}.\mathsf{FunBody} \\
\mathsf{Stack} & ::= & [] \mid \mathsf{Exp} \rightsquigarrow \mathsf{Stack} \\
\mathsf{Cfg} & ::= & \langle \mathsf{Stack}, \mathsf{Env}, \mathsf{Funcs} \rangle \\
\mathsf{Env} & ::= & \mathsf{Array}\{\mathsf{Int}\}\{\mathsf{Int}\} \\
\mathsf{Funcs} & ::= & \mathsf{Array}\{\mathsf{Id}\}\{\mathsf{FunBody}\}
\end{array}
$$

- Sorts are the nonterminals.
- Language constructs are operations.
- The distinguished sort Cfg is the sort of *program configurations*.

Operational Semantics (in Frame Stack Style) (1/2)

$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle$ if $\neg val(e)$ — *assignment*

$\langle i \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle \longrightarrow \langle x{:=}i \rightsquigarrow es, env, fs\rangle$

$\langle x{:=}i \rightsquigarrow es, env, fs\rangle \longrightarrow \langle es, update(env, x, i), fs\rangle$

$\langle x \rightsquigarrow es, env, fs\rangle \longrightarrow \langle lookup(x, env) \rightsquigarrow es, env, fs\rangle$ — *identifier lookup*

$\langle e_1 + e_2 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e_1 \rightsquigarrow \square + e_2 \rightsquigarrow es, env, fs\rangle$ if $\neg val(e_1)$ — *binary operations*

$\langle i_1 \rightsquigarrow \square + e_2 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle i_1 + e_2 \rightsquigarrow es, env, fs\rangle$

$\langle i_1 + e_2 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e_2 \rightsquigarrow i_1 + \square \rightsquigarrow es, env, fs\rangle$ if $\neg val(e_2)$

$\langle i_2 \rightsquigarrow i_1 + \square \rightsquigarrow es, env, fs\rangle \longrightarrow \langle i_1 + i_2 \rightsquigarrow es, env, fs\rangle$

$\langle i_1 + i_2 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle i_1 + i_2 \rightsquigarrow es, env, fs\rangle$

$\langle not\ e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow not\ \square \rightsquigarrow es, env, fs\rangle$ if $\neg val(e)$ — *unary operations*

$\langle b \rightsquigarrow not\ \square \rightsquigarrow es, env, fs\rangle \longrightarrow \langle not\ b \rightsquigarrow es, env, fs\rangle$

$\langle not\ b \rightsquigarrow es, env, fs\rangle \longrightarrow \langle \overline{b} \rightsquigarrow es, env, fs\rangle$

$\langle if\ \top\ then\ e_2\ else\ e_3 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e_2 \rightsquigarrow es, env, fs\rangle$ — *if-then-else*

$\langle if\ \bot\ then\ e_2\ else\ e_3 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e_3 \rightsquigarrow es, env, fs\rangle$

$\langle if\ e_1\ then\ e_2\ else\ e_3 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e_1 \rightsquigarrow if\ \square\ then\ e_2\ else\ e_3 \rightsquigarrow es, env, fs\rangle$ if $\neg val(e_1)$

$\langle b \rightsquigarrow if\ \square\ then\ e_2\ else\ e_3 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle if\ b\ then\ e_2\ else\ e_3 \rightsquigarrow es, env, fs\rangle$

$\langle while\ e_1\ do\ e_2 \rightsquigarrow es, env, fs\rangle \longrightarrow$ — *while loop*

$\langle if\ e_1\ then\ (e_2; while\ e_1\ do\ e_2)\ else\ skip \rightsquigarrow es, env, fs\rangle$

$\langle e_1;e_2 \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e_1 \rightsquigarrow e_2 \rightsquigarrow es, env, fs\rangle$ — *sequence*

$\langle skip \rightsquigarrow es, env, fs\rangle \longrightarrow \langle es, env, fs\rangle$ — *skip*

Operational Semantics (in Frame Stack Style) (2/2)

$$\langle \mathsf{call}\ \mathsf{f} \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \longrightarrow \langle \mathsf{lookup}(\mathsf{f}, \mathsf{fs}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \qquad \textit{function calls}$$

$$\langle \mathsf{call}\ \mathsf{f}(\mathsf{e}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \longrightarrow \langle \mathsf{call}\ \mathsf{f} \rightsquigarrow \mathsf{call}\ \square(\mathsf{e}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle$$

$$\langle \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb} \rightsquigarrow \mathsf{call}\ \square(\mathsf{e}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \longrightarrow \langle \mathsf{call}\ \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb}(\mathsf{e}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle$$

$$\langle \mathsf{call}\ \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb}(\mathsf{e}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \longrightarrow \langle \mathsf{e} \rightsquigarrow \mathsf{call}\ \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb}(\square) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \text{ if } \neg\mathsf{val}(\mathsf{e})$$

$$\langle \mathsf{i} \rightsquigarrow \mathsf{call}\ \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb}(\square) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \longrightarrow \langle \mathsf{call}\ \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb}(\mathsf{i}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle$$

$$\langle \mathsf{call}\ \boldsymbol{\lambda}\mathsf{x}.\mathsf{fb}(\mathsf{i}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle \longrightarrow \langle \mathsf{subst}(\mathsf{x}, \mathsf{i}, \mathsf{fb}) \rightsquigarrow \mathsf{es}, \mathsf{env}, \mathsf{fs}\rangle$$

Example Execution

$\langle [x{:=}x+2], env, fs \rangle \longrightarrow$

Example Execution

$\langle [x{:=}x + 2], env, fs \rangle \longrightarrow$
$\langle [x + 2, x{:=}\square], env, fs \rangle \longrightarrow$

Example Execution

$\langle [x{:=}x + 2], env, fs \rangle \longrightarrow$
$\langle [x + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [x, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$

Example Execution

$\langle [x{:=}x + 2], env, fs \rangle \longrightarrow$
$\langle [x + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [x, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [12, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$

Example Execution

$\langle [x{:=}x + 2], env, fs \rangle \longrightarrow$
$\langle [x + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [x, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [12, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [12 + 2, x{:=}\square], env, fs \rangle \longrightarrow$

Example Execution

$\langle [x{:=}x + 2], env, fs \rangle \longrightarrow$
$\langle [x + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [x, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [12, \square + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [12 + 2, x{:=}\square], env, fs \rangle \longrightarrow$
$\langle [14, x{:=}\square], env, fs \rangle \longrightarrow$

Example Execution

$\langle [x{:=}x + 2], env, fs\rangle \longrightarrow$
$\langle [x + 2, x{:=}\square], env, fs\rangle \longrightarrow$
$\langle [x, \square + 2, x{:=}\square], env, fs\rangle \longrightarrow$
$\langle [12, \square + 2, x{:=}\square], env, fs\rangle \longrightarrow$
$\langle [12 + 2, x{:=}\square], env, fs\rangle \longrightarrow$
$\langle [14, x{:=}\square], env, fs\rangle \longrightarrow$
$\langle [x{:=}14], env, fs\rangle \longrightarrow$

Example Execution

$\langle [\mathrm{x}{:=}\mathrm{x}+2], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [\mathrm{x}+2, \mathrm{x}{:=}\Box], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [\mathrm{x}, \Box+2, \mathrm{x}{:=}\Box], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [12, \Box+2, \mathrm{x}{:=}\Box], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [12+2, \mathrm{x}{:=}\Box], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [14, \mathrm{x}{:=}\Box], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [\mathrm{x}{:=}14], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\langle [], \mathrm{update}(\mathit{env}, \mathrm{x}, 14), \mathit{fs}\rangle \not\longrightarrow .$

Outline

Defining Program Equivalence. The Case for Base Cases

A difficulty in defining program equivalence in a language parametric setting is that different languages may store the *output* (the result) in different places:

Defining Program Equivalence. The Case for Base Cases

A difficulty in defining program equivalence in a language parametric setting is that different languages may store the *output* (the result) in different places:

$\langle [\mathsf{s}{:=}\mathsf{f}(5) + 2], \mathit{env}, \mathit{fs}\rangle \longrightarrow$

$\vdots$

$\langle [], \mathsf{update}(\mathit{env}, \mathsf{s}, 42), \mathit{fs}\rangle \not\longrightarrow.$

$\langle \mathsf{let}\ \mathsf{x} = \mathsf{f}(5)\ \mathsf{in}\ \mathsf{x} + 2\rangle \longrightarrow$

$\vdots$

$42 \not\longrightarrow.$

Defining Program Equivalence. The Case for Base Cases

A difficulty in defining program equivalence in a language parametric setting is that different languages may store the *output* (the result) in different places:

$\langle [\mathsf{s}{:=}\mathsf{f}(5) + 2], \mathit{env}, \mathit{fs}\rangle \longrightarrow$
$\vdots$
$\langle [], \mathsf{update}(\mathit{env}, \mathsf{s}, 42), \mathit{fs}\rangle \not\longrightarrow .$

$\langle \mathsf{let}\ \mathsf{x} = \mathsf{f}(5)\ \mathsf{in}\ \mathsf{x} + 2\rangle \longrightarrow$
$\vdots$
$42 \not\longrightarrow .$

We parametrize in the set of base case: we let $\mathbb{B}$ be pairs of terminal configurations that are considered equivalent.

Simulations between Program Configurations

Given: Two operational semantics $\mathcal{R}_L$ and $\mathcal{R}_R$; the sorts CfgL and CfgR.

Definition (Full (Partial) Simulation)

A symbolic program configuration P *is fully (partially) simulated by a symbolic program configuration* Q *under constraint* ϕ *with a set of base cases* $\mathbb{B}$, written

$$\mathbb{B} \vDash P \prec Q \text{ if } \phi \qquad (\mathbb{B} \vDash P \preceq Q \text{ if } \phi),$$

if, for any valuation ρ such that $\rho(\phi) = \top$ and for any complete path $\rho(P) \longrightarrow^*_{\mathcal{R}_L} \mathrm{P}'$, there exists a complete path $\rho(Q) \longrightarrow^*_{\mathcal{R}_R} \mathrm{Q}'$ such that $(\mathrm{P}', \mathrm{Q}') \in \mathbb{B}$ (or – for partial simulation only – there exists an infinite path $\rho(Q) \longrightarrow_{\mathcal{R}_R} \ldots$);

Program Equivalence

Definition (Full (Partial) Equivalence)

Two symbolic program configurations P and Q are *fully equivalent (partially equivalent)* under constraint ϕ with a set of base cases $\mathbb{B}$, written

$$\mathbb{B} \vDash P \sim Q \text{ if } \phi \qquad (\mathbb{B} \vDash P \simeq Q \text{ if } \phi),$$

if $\mathbb{B} \vDash P \prec Q$ if ϕ *and* $\mathbb{B}^{-1} \vDash Q \prec P$ if ϕ ($\mathbb{B} \vDash P \preceq Q$ if ϕ *and* $\mathbb{B}^{-1} \vDash Q \preceq P$ if ϕ).

Outline

Basic Idea

We prove equivalence by two-sided simulation.

Basic Idea

We prove equivalence by two-sided simulation.
To prove $P \leftrightarrows Q$ if ϕ, we construct a set G of goals such that:

1. $P \leftrightarrows Q$ if $\phi \in G$;
2. G may contain other helper goals (lemmas).

Basic Idea

We prove equivalence by two-sided simulation.
To prove $P \mathrel{\therefore\!\prec} Q$ if ϕ, we construct a set G of goals such that:

1. $P \mathrel{\therefore\!\prec} Q$ if $\phi \in G$;
2. G may contain other helper goals (lemmas).

We provide a sound proof system:

Theorem (Soundness for full/partial simulation)

If $G, B \vdash^0 G$ *and* $[\![B]\!] \subseteq \mathbb{B}$, *then for any simulation formula*

$$P \mathrel{\therefore\!\prec} Q \text{ if } \phi \in G,$$

we have that

$$\mathbb{B} \vDash P \mathrel{\therefore\!\prec} Q \text{ if } \phi.$$

Proof System for Simulation

Notation: $sub\Big((P,Q),R\Big) \triangleq \bigvee_{P' \precsim Q' \text{ if } \phi' \in R} \exists var(P',Q',\phi').(\phi' \wedge P{=}P' \wedge Q{=}Q')$

$$\textsc{Axiom}\ \frac{}{G,B \vdash^{g} P \precsim Q \text{ if } \bot}$$

$$\textsc{Base}\ \frac{G,B \vdash^{g} P \precsim Q \text{ if } \phi \wedge \neg\phi_B}{G,B \vdash^{g} P \precsim Q \text{ if } \phi} \models \phi_B \rightarrow \bigvee_{Q' \text{ if } \phi' \in \Delta^{\leq k}_{\overline{\mathcal{R}}_R}(Q)} \phi' \rightarrow sub\Big((P,Q'),B\Big)$$

$$\textsc{Circ}^{\prec}\ \frac{G,B \vdash^{1} P \prec Q \text{ if } \phi \wedge \neg\phi_G}{G,B \vdash^{1} P \prec Q \text{ if } \phi} \models \phi_G \rightarrow \bigvee_{Q' \text{ if } \phi' \in \Delta^{\leq k}_{\overline{\mathcal{R}}_R}(Q)} \phi' \rightarrow sub\Big((P,Q'),G\Big)$$

$$\textsc{Circ}^{\preceq}\ \frac{G,B \vdash^{g} P \preceq Q \text{ if } \phi \wedge \neg\phi_G}{G,B \vdash^{g} P \preceq Q \text{ if } \phi} \models \phi_G \rightarrow \bigvee_{Q' \text{ if } \phi' \in \Delta^{\geq 1-g,\leq k}_{\overline{\mathcal{R}}_R}(Q)} \phi' \rightarrow sub\Big((P,Q'),G\Big)$$

$$\textsc{Step}\ \frac{G,B \vdash^{1} P^i \precsim Q \text{ if } \phi^i \text{ (for all } 1 \leq i \leq n)}{G,B \vdash^{g} P \precsim Q \text{ if } \phi \wedge \neg\phi^1 \wedge \ldots \wedge \neg\phi^n}\ \Delta_{\mathcal{R}_L}(P \text{ if } \phi){=}\{P^i \text{ if } \phi^i \mid 1 \leq i \leq n\}$$

Outline

Setting Resource Bounds

Let us assume that we are interested in interpreting `WH` programs on a finite-stack machine.

Setting Resource Bounds

Let us assume that we are interested in interpreting `WH` programs on a finite-stack machine.
Initial rewrite rule:

$$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle \text{ if } \neg val(e)$$

Setting Resource Bounds

Let us assume that we are interested in interpreting `WH` programs on a finite-stack machine.
Initial rewrite rule:

$$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle \text{ if } \neg val(e)$$

Updated rewrite rule:

$$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle \text{ if } \neg val(e) \wedge \underbrace{len(es) < k}_{\textit{new constraint}}$$

Setting Resource Bounds

Let us assume that we are interested in interpreting WH programs on a finite-stack machine.
Initial rewrite rule:

$$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle \text{ if } \neg val(e)$$

Updated rewrite rule:

$$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\square \rightsquigarrow es, env, fs\rangle \text{ if } \neg val(e) \wedge \underbrace{len(es) < k}_{\textit{new constraint}}$$

Call the initial language $\mathtt{WH}_1$ and the finite-stack language $\mathtt{WH}_2$.

Equivalence in the Presence of Resource Bounds

We study using our method the equivalence in $\mathtt{WH}_1$ and $\mathtt{WH}_2$ of the programs:

$$\langle [\text{call f(N)}], \text{env}, \mathit{fs}\rangle \qquad \text{and}$$
$$\langle [\text{call F(N, 0, 0)}], \text{env}, \mathit{fs}\rangle$$

for $\text{N} \geq 0$, where the recursive functions f and F are defined by the function map

$$\mathit{fs} = \{\ \text{f} \mapsto \boldsymbol{\lambda}\text{n. if n} = 0 \text{ then 0 else n} + \text{call f(n} - 1),$$
$$\text{F} \mapsto \boldsymbol{\lambda}\text{n.}\boldsymbol{\lambda}\text{i.}\boldsymbol{\lambda}\text{a. if i} \leq \text{n then call F(n, i} + 1, \text{a} + \text{i) else a}\}.$$

The functions f and F

exp. phase

$$\langle [f(3)] \rangle \longrightarrow^*$$
$$\langle [f(2), 3 + \Box] \rangle \longrightarrow^*$$
$$\langle [f(1), 2 + \Box, 3 + \Box] \rangle \longrightarrow^*$$
$$\langle [f(0), 1 + \Box, 2 + \Box, 3 + \Box] \rangle \longrightarrow^*$$

contr. phase

$$\langle [0, 1 + \Box, 2 + \Box, 3 + \Box] \rangle \longrightarrow^*$$
$$\langle [1, 2 + \Box, 3 + \Box] \rangle \longrightarrow^*$$
$$\langle [3, 3 + \Box] \rangle \longrightarrow^*$$
$$\langle [6] \rangle \not\longrightarrow$$

$$\langle [F(3, 0, 0)] \rangle \longrightarrow^*$$
$$\langle [F(3, 1, 0)] \rangle \longrightarrow^*$$
$$\langle [F(3, 2, 1)] \rangle \longrightarrow^*$$
$$\langle [F(3, 3, 3)] \rangle \longrightarrow^*$$
$$\langle [F(3, 4, 6)] \rangle \longrightarrow^*$$
$$\langle [6] \rangle \not\longrightarrow$$

Equivalence in the Presence of Resource Bounds

We let $\mathbb{B} = \{(\langle [i], env, fs \rangle, \langle [i'], env', fs' \rangle) \mid i = i' \wedge i, i' \in \mathbb{Z}\}$.

Equivalence in the Presence of Resource Bounds

We let $\mathbb{B} = \{(\langle [i], env, fs\rangle, \langle [i'], env', fs'\rangle) \mid i = i' \wedge i, i' \in \mathbb{Z}\}$.
$G = \{$

$$
\begin{array}{lcll}
\langle [\text{call f(N)}], \text{env}, fs\rangle & \mathrel{:\!\asymp} & \langle [\text{call F(N, 0, 0)}], \text{env}, fs\rangle & \text{if} \quad 0 \leq \text{N}, \\
\langle \text{call f(I}-1) \rightsquigarrow \text{reduce(I, N)}, \text{env}, fs\rangle & \mathrel{:\!\asymp} & \langle [\text{call F(N, 0, 0)}], \text{env}, fs\rangle & \text{if} \quad 0 \leq \text{I} \leq \text{N}, \\
\langle \text{S} \rightsquigarrow \text{reduce(I, N)}, \text{env}, fs\rangle & \mathrel{:\!\asymp} & \langle [\text{call F(N, I, S)}], \text{env}, fs\rangle & \text{if} \quad 1 \leq \text{I} \leq \text{N}\},
\end{array}
$$

where

$$
\begin{array}{l}
\text{reduce(i, n)} \longrightarrow [] \text{ if } \text{i} > \text{n}, \\
\text{reduce(i, n)} \longrightarrow (\text{i} + \square) \rightsquigarrow \text{reduce(i}+1, \text{n}) \text{ if } \text{i} \leq \text{n}.
\end{array}
$$

Equivalence in the Presence of Resource Bounds

We let $\mathbb{B} = \{(\langle [i], env, fs\rangle, \langle [i'], env', fs'\rangle) \mid i = i' \wedge i, i' \in \mathbb{Z}\}$.
$G = \{$

$$
\begin{array}{lcll}
\langle [\text{call f(N)}], \text{env}, fs\rangle & \rightarrowtail & \langle [\text{call F(N, 0, 0)}], \text{env}, fs\rangle & \text{if} \quad 0 \leq \text{N}, \\
\langle \text{call f(I}-1) \rightsquigarrow \text{reduce(I, N)}, \text{env}, fs\rangle & \rightarrowtail & \langle [\text{call F(N, 0, 0)}], \text{env}, fs\rangle & \text{if} \quad 0 \leq \text{I} \leq \text{N}, \\
\langle \text{S} \rightsquigarrow \text{reduce(I, N)}, \text{env}, fs\rangle & \rightarrowtail & \langle [\text{call F(N, I, S)}], \text{env}, fs\rangle & \text{if} \quad 1 \leq \text{I} \leq \text{N}\},
\end{array}
$$

where

$$
\begin{array}{l}
\text{reduce(i, n)} \longrightarrow [] \text{ if i}>\text{n}, \\
\text{reduce(i, n)} \longrightarrow (\text{i} + \square) \rightsquigarrow \text{reduce(i}+1, \text{n}) \text{ if i}\leq\text{n}.
\end{array}
$$

Using the sets G, B defined above, we have that

$$G, B \vdash^0 G \text{ for partial and full simulation}$$

and that

$$G^{-1}, B^{-1} \vdash^0 G^{-1} \text{ for partial simulation.}$$

Our proof system cannot show $G^{-1}, B^{-1} \vdash^0 G^{-1}$ for technical reasons.

Equivalence in the Presence of Resource Bounds

We let $\mathbb{B} = \{(\langle [i], env, fs\rangle, \langle [i'], env', fs'\rangle) \mid i = i' \wedge i, i' \in \mathbb{Z}\}$.
$G = \{$

$$
\begin{array}{lcll}
\langle [\text{call f(N)}], \text{env}, fs\rangle & \mathrel{:\!\asymp} & \langle [\text{call F(N, 0, 0)}], \text{env}, fs\rangle & \text{if} \quad 0 \leq \text{N}, \\
\langle \text{call f(I}-1) \rightsquigarrow \text{reduce(I, N)}, \text{env}, fs\rangle & \mathrel{:\!\asymp} & \langle [\text{call F(N, 0, 0)}], \text{env}, fs\rangle & \text{if} \quad 0 \leq \text{I} \leq \text{N}, \\
\langle \text{S} \rightsquigarrow \text{reduce(I, N)}, \text{env}, fs\rangle & \mathrel{:\!\asymp} & \langle [\text{call F(N, I, S)}], \text{env}, fs\rangle & \text{if} \quad 1 \leq \text{I} \leq \text{N}\},
\end{array}
$$

where

$$
\begin{array}{l}
\text{reduce(i, n)} \longrightarrow [] \text{ if i}>\text{n}, \\
\text{reduce(i, n)} \longrightarrow (\text{i} + \square) \rightsquigarrow \text{reduce(i}+1, \text{n}) \text{ if i}\leq\text{n}.
\end{array}
$$

Using the sets G, B defined above, we have that

$$G, B \vdash^0 G \text{ for partial and full simulation}$$

and that

$$G^{-1}, B^{-1} \vdash^0 G^{-1} \text{ for partial simulation.}$$

Our proof system cannot show $G^{-1}, B^{-1} \vdash^0 G^{-1}$ for technical reasons.
So we prove partial equivalence between f and F, but only part of the full equivalence between f and F.

Outline

Implementation

Prototype implementation of the two proof systems in the RMT tool at:

http://profs.info.uaic.ro/~stefan.ciobaca/wpte2021.

Example	Ctrl	RMT
fib02	4.02s	25.73s
fib03	3.06s	27.55s
fib04	Timeout	77.40s
fib05	3.99s	36.67s
fib06	32.81s	63.39s
fib07	2.74s	59.31s
fib08	3.44s	60.77s
fib09	3.09s	41.51s
fib10	2.34s	35.91s
fib11	Timeout	79.52s
fib12	No	No
sum01	2.39s	10.51s
sum02	2.33s	12.36s
sum03	2.62s	12.30s

Optimization	PEC	CORK		RMT
Code hoisting	✓	0.32s		0.41s
Constant propagation	✓	0.33s		0.31s
Copy propagation	✓	0.33s		0.26s
If-conversion	✓	0.34s		0.48s
Partial redundancy elimination	✓	0.34s		0.75s
Loop invariant code motion	✓	3.48s		3.79s
Loop peeling	✓	3.26s		0.97s
Loop unrolling	✓	12.17s		7.09s
Loop unswitching	✓	8.19s		4.71s
Software pipelining	✓	8.02s		3.56s
Loop fission	✓	23.45s	*	10.40s
Loop fusion	✓	23.34s	*	9.67s
Loop interchange	✓	29.30s	*	108.63s
Loop reversal	✓	8.41s		2.70s
Loop skewing	✓	8.50s		7.68s
Loop flattening	×	×	*	8.14s
Loop strength reduction	×	5.63s		5.26s
Loop tiling 01	×	10.94s		25.41s
Loop tiling 02			*	21.58s

Conclusion

1. Promising approach to operationally-based equivalence.
2. Requires extending LCTRSs with *axiomatized symbols* (like reduce), which raise new research questions (unification modulo axiomatized symbols).
3. Our approach allows for nondeterminism in the definitions and proofs of equivalence, and we will exploit this in future work.

Thank you!