Modular Bisimulation Theory for Computations and Values

Martin Churchill and Peter D. Mosses
Swansea University, UK

FoSSaCS, Rome
March 2013

Part of the project:

PLanCompS

http://www.plancomps.org

EPSRC-funded, 2011-2015

{Swansea, Royal Holloway, City}, UK

Scaling up formal semantics to real-world programming languages

Upcoming workshop:
Scalable Language Specification (SLS13)
with project partner Microsoft Research Cambridge

- Here we work on equivalence for operational semantics:
  - bisimulation congruence formats

Bisimulation

- **Bisimilarity** provides notions of behavioural equivalence for operational semantics.
- Defined for **transition relation** $s \xrightarrow{\ell} t$.
- If s and t are (strongly) bisimilar, they can match each step and remain bisimilar:

$$s \approx t \text{ and } s \xrightarrow{\ell} s' \text{ implies } \exists t' \text{ with } t \xrightarrow{\ell} t' \text{ and } s' \approx t'.$$

Congruence

- Any notion of equivalence should be a **congruence**:

  For each f, $s_1 \approx t_1$, ..., $s_n \approx t_n$ implies
  $$f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)$$

  'one can replace a term by an equivalent term in a larger context, and the overall context remains equivalent'
- Enables **compositional reasoning**

But:

- Bisimilarity is **not** guaranteed to be a congruence!

Congruence formats

- Transition systems for operational semantics can be defined inductively by **SOS rules** (Plotkin)
- There are known **formats** for such rules which guarantee that bisimulation is a congruence. (GSOS, tyft/tyxt, ...)

$$\frac{\{t_i \xrightarrow{\ell_i} y_i : i \in I\}}{f(x_1, \ldots, x_n) \xrightarrow{\ell} t}$$

$$\frac{x \xrightarrow{\ell} x'}{x \| y \xrightarrow{\ell} x' \| y} \qquad \frac{y \xrightarrow{\ell} y'}{x \| y \xrightarrow{\ell} x \| y'}$$

- Well-suited to **process algebras** but not to **programming languages**:
  - Programming language terms can **compute values**
  - Programming language transition systems generally have **auxiliary entities** (stores, environments, ...) which can contain arbitrary terms

- Well-suited to **process algebras** but not to **programming languages**:
  - Programming language terms can **compute values**
  - Programming language transition systems generally have **auxiliary entities** (stores, environments, ...) which can contain arbitrary terms

$$\frac{\rho[x \mapsto v] \vdash s \to s'}{\rho \vdash \textsf{apply}(\lambda x.s, v) \to \textsf{apply}(\lambda x.s', v)}$$

$$\rho \vdash \textsf{apply}(\lambda x.v_1, v_2) \to v_1$$

$\Rightarrow$ not in congruence formats!

Contribution

What about bisimulation congruence for programming languages?

In this work we describe a new congruence format which:

- can deal adequately with computed values
- can deal adequately with auxiliary entities
  - $\Rightarrow$ **higher-order** bisimulation
- scales up to **real programming languages** *(supports MSOS)*

Bisimulation for Value-computation Systems

Values and Computations

We distinguish between value terms and computational terms.
"Values are, computations do" *(Levy)*

Values:

- Booleans, integers, function abstractions, ...

Computations:

- Expressions, commands, declarations, processes, programs, ...

Values and computations

Values can be *inspected*, computations have *behaviour*.

- Any observing context must be able to distinguish values e.g. true and false.
- But they cannot interrogate the structure of arbitrary computational terms:
  - otherwise equivalence would reduce to syntactic identity.

Values and computations

Values can be *inspected*, computations have *behaviour*.

- Any observing context must be able to distinguish values e.g. true and false.
- But they cannot interrogate the structure of arbitrary computational terms:
  - otherwise equivalence would reduce to syntactic identity.

Algebraic signature, values determined by a set of *value constructors*

(true, false, thunk(−), ...)

Value-computation Bisimulation

A symmetric relation R satisfying the usual bisimulation step conditions, plus:

- If $v(s_1, \ldots, s_n)\ R\ v_1$ with $v \in VC$, then $v_1 = v(t_1, \ldots, t_n)$ with $s_i\ R\ t_i$ for $1 \leq i \leq n$.

Bisimilar values have the same head constructor and bisimilar arguments.

Value-computation Bisimulation

A symmetric relation R satisfying the usual bisimulation step conditions, plus:

- If $v(s_1, \ldots, s_n)\ R\ v_1$ with $v \in VC$, then $v_1 = v(t_1, \ldots, t_n)$ with $s_i\ R\ t_i$ for $1 \leq i \leq n$.

Bisimilar values have the same head constructor and bisimilar arguments.

Congruence format:

- Arguments of conclusion source, targets of premises must be *patterns* — a term made of variables and value constructors.
- Generalisation of *tyft* format (Groote, Vaandrager)

$$\frac{\{t_i \xrightarrow{\ell_i} u_i : i \in I\}}{f(w_1, \ldots, w_n) \xrightarrow{\ell} t}$$

Higher-order Bisimulation for MSOS

Auxiliary entities

- For programming languages, we can place *auxiliary entities* (store etc) in the label – indexed record of terms.

Examples:

- **env**, an environment mapping identifiers to the values they are bound to ('read' component)
- **exc**, which signals whether an exception has occurred, and if so which ('write' component)
- **store**, a mapping from reference cells to their current value (changeable – 'read' + 'write' component)
- ...

Examples

$$\mathsf{bound}(x) \xrightarrow{\{\mathbf{env}=\rho,-\}} \mathsf{lookup}(\rho, x)$$

$$\frac{y \xrightarrow{\{\mathbf{exc}'=\mathsf{nil},\ldots\}} y'}{\mathsf{catch}(y, f) \xrightarrow{\{\mathbf{exc}'=\mathsf{nil},\ldots\}} \mathsf{catch}(y', f)}$$

$$\frac{y \xrightarrow{\{\mathbf{env}=\mathsf{update}(\rho,x,v),\ldots\}} y'}{\mathsf{let}(x, v, y) \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \mathsf{let}(x, v, y')}$$

$$\frac{y \xrightarrow{\{\mathbf{exc}'=\mathsf{exc}(e),\ldots\}} y'}{\mathsf{catch}(y, f) \xrightarrow{\{\mathbf{exc}'=\mathsf{nil},\ldots\}} \mathsf{apply}(f, e)}$$

$$\mathsf{throw}(e) \xrightarrow{\{\mathbf{exc}'=\mathsf{exc}(e),-\}} \mathsf{stuck}$$

$$\mathsf{catch}(v, f) \xrightarrow{\{-\}} v$$

'...' is a variable ranging over the 'rest of the label'

'–' provides defaults for unmentioned entities

- lookup is a computational constructor for maps.
- update, nil, exc are value constructors.
- NB: information flows between source, label components, target

Higher-order bisimulation

We wish to ensure that bisimulation is a congruence.

If

$$s \approx t \text{ is to imply } \mathsf{throw}(s) \approx \mathsf{throw}(t)$$

we must allow label components to vary up to bisimulation in the 'step' (a higher-order bisimulation).

Bisimulation for MSOS

As before, except step condition becomes:

- If $s\ R\ t$ and $s \xrightarrow{\ell} s'$ then $\exists t', \ell'$ with $s'\ R\ t'$, $t \xrightarrow{\ell'} t'$, reads(ℓ') = reads(ℓ) and writes$(\ell)\ R$ writes(ℓ').

Write components in the label can vary up to bisimulation

Bisimulation for MSOS

As before, except step condition becomes:

- If $s\ R\ t$ and $s \xrightarrow{\ell} s'$ then $\exists t', \ell'$ with $s'\ R\ t'$, $t \xrightarrow{\ell'} t'$, $\mathrm{reads}(\ell') = \mathrm{reads}(\ell)$ and $\mathrm{writes}(\ell)\ R\ \mathrm{writes}(\ell')$.

Write components in the label can vary up to bisimulation

Congruence format: (MSOS-tyft)

- In the conclusion, readable components must be patterns
- In premises, writeable components must be patterns

Bisimulation for MSOS

As before, except step condition becomes:

- If $s\ R\ t$ and $s \xrightarrow{\ell} s'$ then $\exists t', \ell'$ with $s'\ R\ t'$, $t \xrightarrow{\ell'} t'$, $\text{reads}(\ell') = \text{reads}(\ell)$ and $\text{writes}(\ell)\ R\ \text{writes}(\ell')$.

Write components in the label can vary up to bisimulation

Congruence format: (MSOS-tyft)

- In the conclusion, readable components must be patterns
- In premises, writeable components must be patterns

Requires lemma:

- *'One can replace inputs (read components) for bisimilar ones to yield bisimilar outputs (write components)'*

$\sim$ applicative bisimulation (Howe)

Conclusions

Expressivity

MSOS-tyft format expressive enough to model Caml Light:

- Higher-order functions, imperative state, exceptions, mutual recursion, pattern matching, . . .

$\Rightarrow$ Equivalences are valid in *arbitrary program contexts*

Expressivity

MSOS-tyft format expressive enough to model Caml Light:

- Higher-order functions, imperative state, exceptions, mutual recursion, pattern matching, . . .

⇒ Equivalences are valid in *arbitrary program contexts*

[We translate Caml Light programs into reusable constructs, yielding a *component-based formal semantics* of the language.]

Further Directions

Theory:

- Multisorted signatures
- Bisimulations parametrised on read components (**env**, . . .)
  cf. state-based bisimilarity (Mousavi)
- Negative premises

Practice:

- Larger language examples, e.g. C#, Java, . . .

Conclusions

We have:

- defined a bisimulation congruence format dealing with **computed values** and **auxiliary entities**
- which supports a **higher-order** notion of bisimulation and Modular SOS
- and is **expressive enough** to treat a real world programming language.

Thank You.

APPENDIX

(some more detailed slides)

Value-computation signatures

Definition

A *value-computation signature* consists of a set of constructors C, each with an arity in $\mathbb{N}$, and a set of *value constructors* $VC \subseteq C$. We let T denote the set of terms, and $V \subseteq T$ the set of *value terms* whose outermost constructor is in VC.

- Constants like true,5,... are nullary value constructors, and hence values.
- cond is not a value constructor, so cond(B, C, D) is never a value.
- thunk is a value constructor that can wrap an arbitrary computation into a value.

Value-computation transition systems

- Fix a set of labels L.
- We wish to define our relation $s \xrightarrow{l} t$ with $s, t \in T$ and $l \in L$ to model computations. We define this via inductive rules.
- It is useful to introduce an additional internal relation for silent, context-insensitive steps which we write as $\Rightarrow$. This relation is reflexive, transitive and a precongruence.

$$x \Rightarrow x \qquad \frac{x_1 \Rightarrow y_1 \qquad \cdots \qquad x_n \Rightarrow y_n}{f(x_1, \ldots, y_n) \Rightarrow f(x_1, \ldots, y_n)}$$

$$\frac{x \Rightarrow y \qquad y \Rightarrow z}{x \Rightarrow z} \qquad \frac{x \Rightarrow x_1 \qquad x_1 \xrightarrow{l} y_1 \qquad y_1 \Rightarrow y}{x \xrightarrow{l} y}$$

Value-computation Bisimulation

Definition

A *value-computation bisimulation* over a given value-computation transition system $(\Sigma, L, \rightarrow, \Rightarrow)$ is a symmetric relation $R \subseteq T_\Sigma \times T_\Sigma$ such that

- If $s\ R\ t$ and $s \xrightarrow{l} s'$ then $\exists t'$ with $s'\ R\ t'$ and $t \xrightarrow{l} t'$.
- If $s\ R\ t$ and $s \Rightarrow s'$ then $\exists t'$ with $s'\ R\ t'$ and $t \Rightarrow t'$.
- If $v(s_1, \ldots, s_n)\ R\ t$ with $v \in VC$, then $t \Rightarrow v(t_1, \ldots, t_n)$ with $s_i\ R\ t_i$ for $1 \leq i \leq n$.

Two terms s and t are *value-computation bisimilar*, written $s \approx_{vc} t$, if there exists a value-computation bisimulation R with $s\ R\ t$.

Congruence Format

$$\frac{\{s_i \rightsquigarrow_i u_i : i \in I\}}{f(w_1, \ldots, w_n) \rightsquigarrow t}$$

(Each $\rightsquigarrow, \rightsquigarrow_i \in \{\Rightarrow\} \cup \{\xrightarrow{a} : a \in L\}$.)

- Adaptation of the *tyft* format [**?**]
- But u_i and w_i generalised from variables to *patterns* — a term made of variables and value constructors.
- Allows rules such as $\overline{\mathsf{seq}(\mathsf{skip}, s) \Rightarrow s}$ and $\overline{\mathsf{force}(\mathsf{thunk}(s)) \Rightarrow s}$.

For rules in this *value-added tyft* format, vc-bisimilarity is a congruence.

Composition and Unobservability

For each label component, there are default 'unobservable' labels − for unmentioned components.

$$\mathsf{print}(x) \xrightarrow{\{\mathbf{output}'=x,\mathbf{env}=\rho,\mathbf{store}=\sigma,\mathbf{store}'=\sigma,\mathbf{exc}=\mathsf{nil}\}} \mathsf{skip}$$

Labels may also be *composed*:

$$\mathsf{seq}(\mathsf{assign}(\mathsf{y},6),\mathsf{assign}(\mathsf{x},5)) \xrightarrow{\{\mathbf{store}=\sigma,\mathbf{store}'=\mathsf{update}(\sigma,\mathsf{y},6),-\}} \mathsf{seq}(\mathsf{skip},\mathsf{assign}(\mathsf{x},5)) \Rightarrow$$
$$\mathsf{assign}(\mathsf{x},5) \xrightarrow{\{\mathbf{store}=\sigma_1,\mathbf{store}'=\mathsf{update}(\sigma_1,\mathsf{x},5),-\}} \mathsf{skip}$$

$$\mathsf{seq}(\mathsf{assign}(\mathsf{y},6),\mathsf{assign}(\mathsf{x},5)) \xrightarrow{\{\mathbf{store}=\sigma,\mathbf{store}'=\sigma[\mathsf{x}\mapsto 6,\mathsf{y}\mapsto 5],-\}} *\mathsf{skip}$$

- Ensures e.g. *single-threaded store*.
- Each label component is a *category*.

Bisimulation for MSOS

Definition

Given a value-computation transition system $(\Sigma, \mathcal{L}(T_\Sigma), \rightarrow, \Rightarrow)$ generated from an MSOS specification, an *MSOS bisimulation* is a symmetric relation $R \subseteq T \times T$ such that:

- If $s\ R\ t$ and $s \xrightarrow{L} s'$ then $\exists t', L'$ with $s'\ R\ t'$, $t \xrightarrow{L'} t'$, $\text{reads}(L') = \text{reads}(L)$ and $\text{writes}(L)\ R\ \text{writes}(L')$.
- If $s\ R\ t$ and $s \Rightarrow s'$ then $\exists t'$ with $s'\ R\ t'$ and $t \Rightarrow t'$.
- If $v(s_1, \ldots, s_n)\ R\ t$ with $v \in VC$, then $t \Rightarrow v(t_1, \ldots, t_n)$ with $s_i\ R\ t_i$ for $1 \leq i \leq n$.

Two terms s and t are *MSOS bisimilar*, written $s \approx_{msos} t$, if there exists an MSOS bisimulation R with $s\ R\ t$.

MSOS tyft format

$$\frac{\{s_i \rightsquigarrow_i u_i : i \in I\}}{f(w_1, \ldots, w_n) \rightsquigarrow t}$$

(Each $\rightsquigarrow, \rightsquigarrow_i \in \{\Rightarrow\} \cup \{\xrightarrow{L}\}$.)

For labels L:

- In the conclusion $\rightsquigarrow$, readable components must be patterns $\{\mathbf{l} = u, \mathbf{l}' = t, \ldots\}$
- In premises $\rightsquigarrow_i$, writeable components must be patterns $\{\mathbf{l} = t, \mathbf{l}' = u, \ldots\}$
- We allow composition and unobservability in conclusion

Bisimilarity-preservation for unprimed entities

To show congruence of bisimilarity for MSOS tyft, we need:

'One can replace read components (environment, initial store, ...) for bisimilar ones to yield bisimilar write components (final store, thrown exceptions, ...) and target term' $\sim$ applicative bisimulation (Howe)

- If $s \xrightarrow{L} s'$ and $\text{reads}(L) \approx trs$ then:
  $\exists\ s''$, tws such that $s' \approx s''$, $\text{writes}(L) \approx tws$ and $s \xrightarrow{L'} s''$ for $\text{reads}(L') = trs$ and $\text{writes}(L') = tws$.

Theorem
MSOS-bisimilarity is a congruence for the MSOS tyft format.

$$\mathrm{throw}(x) \xrightarrow{\{\mathbf{exc}'=\mathrm{exc}(x),-\}} \mathrm{stuck}$$

$$\frac{x \xrightarrow{\{\mathbf{exc}'=\mathrm{nil},\ldots\}} x'}{\mathrm{catch}(x,y,z) \xrightarrow{\{\mathbf{exc}'=\mathrm{nil},\ldots\}} \mathrm{catch}(x',y,z)}$$

$$\mathrm{catch}(v,x,z) \Rightarrow v$$

$$\frac{x \xrightarrow{\{\mathbf{exc}'=\mathrm{exc}(x_1),\ldots\}} x'}{\mathrm{catch}(x,y,z) \xrightarrow{\{\mathbf{exc}'=\mathrm{nil},\ldots\}} \mathrm{let}(y,x_1,z)}$$

$$\frac{y \xrightarrow{\{\ldots\}} y'}{\mathrm{assign}(x,y) \xrightarrow{\{\ldots\}} \mathrm{assign}(x,y')}$$

$$\mathrm{assign}(x,v) \xrightarrow{\{\mathbf{store}=\sigma,\mathbf{store}'=\mathrm{update}(\sigma,x,v),-\}} \mathrm{skip}$$

$$\mathrm{deref}(x) \xrightarrow{\{\mathbf{store}=\sigma,-\}} \mathrm{lookup}(\sigma,x)$$

- assign and deref are constructors for imperative store.
- throw and catch for exception handling.

NB: '–' provides defaults for the 'rest of the label'

(e.g. $\mathbf{store} = \mathbf{store}'$, $\mathbf{exc}' = \mathrm{nil}$)

Application and Abstraction

$$\frac{x \xrightarrow{\{\ldots\}} x'}{\mathsf{apply}(x, y) \xrightarrow{\{\ldots\}} \mathsf{apply}(x', y)}$$

$$\frac{y \xrightarrow{\{\ldots\}} y'}{\mathsf{apply}(v, y) \xrightarrow{\{\ldots\}} \mathsf{apply}(v, y')}$$

$$\frac{y \xrightarrow{\{\mathbf{env}=\mathsf{update}(\rho,x,v),\ldots\}} y'}{\mathsf{apply}(\mathsf{abs}(x, y, \rho), v) \xrightarrow{\{\mathbf{env}=\rho_1,\ldots\}} \mathsf{apply}(\mathsf{abs}(x, y', \rho), v)}$$

$$\mathsf{apply}(\mathsf{abs}(x, v_1, \rho), v_2) \Rightarrow v_1$$

$$\mathsf{lambda}(x, y) \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \mathsf{abs}(x, y, \rho)$$