Diagrams for Meaning Preservation

2003-06-13

Joe Wells[*] Detlef Plump[†] Fairouz Kamareddine[*]

[*] Heriot-Watt University `www.cee.hw.ac.uk/ultra`

[†] University of York `www.cs.york.ac.uk/~det`

Useful
Logics,
Types,
Rewriting, and their
Automation

Overview

- **Motivation.**
- The AES framework.
- Methods for proving meaning preservation.
- Discussion.

Some Notation

- The notation $t_1 \xrightarrow{R}\!\!\!\!\twoheadrightarrow t_2$ means t_1 is related to t_2 by zero or more steps of the relation R.

Some Notation

- The notation $t_1 \xrightarrow{R}\!\!\!\!\twoheadrightarrow t_2$ means t_1 is related to t_2 by zero or more steps of the relation R.

- The notation $t_1 \xrightarrow{R,\mathsf{nf}}\!\!\!\!\twoheadrightarrow t_2$ means $t_1 \xrightarrow{R}\!\!\!\!\twoheadrightarrow t_2$ and furthermore t_2 is not related by 1 further step of R to any other term.

Some Notation

- The notation $t_1 \xrightarrow{R}\!\!\!\!\!\!\twoheadrightarrow t_2$ means t_1 is related to t_2 by zero or more steps of the relation R.
- The notation $t_1 \xrightarrow{R,\mathsf{nf}}\!\!\!\!\!\!\twoheadrightarrow t_2$ means $t_1 \xrightarrow{R}\!\!\!\!\!\!\twoheadrightarrow t_2$ and furthermore t_2 is not related by 1 further step of R to any other term.
- The notation has-nf(R, t_1) means $t_1 \xrightarrow{R,\mathsf{nf}}\!\!\!\!\!\!\twoheadrightarrow t_2$ for some t_2.

Some Notation

- The notation $t_1 \xrightarrow{R}\!\!\!\!\twoheadrightarrow t_2$ means t_1 is related to t_2 by zero or more steps of the relation R.
- The notation $t_1 \xrightarrow{R,\mathsf{nf}}\!\!\!\!\twoheadrightarrow t_2$ means $t_1 \xrightarrow{R}\!\!\!\!\twoheadrightarrow t_2$ and furthermore t_2 is not related by 1 further step of R to any other term.
- The notation $\mathsf{has\text{-}nf}(R, t_1)$ means $t_1 \xrightarrow{R,\mathsf{nf}}\!\!\!\!\twoheadrightarrow t_2$ for some t_2.
- Trm, Conf, LConf, and Std are short names for *termination*, *confluence local confluence*, and *standardization*.

An Example Operational Semantics

Example: functions with call-by-name evaluation to weak head normal form.

An Example Operational Semantics

Example: functions with call-by-name evaluation to weak head normal form.

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

An Example Operational Semantics

Example: functions with call-by-name evaluation to weak head normal form.

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Evaluation contexts: $E \in \mathsf{EvalContext} ::= \Box \mid (E\, t)$

An Example Operational Semantics

Example: functions with call-by-name evaluation to weak head normal form.

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Evaluation contexts: $E \in \mathsf{EvalContext} ::= \Box \mid (E\, t)$

Evaluation rewriting: $E[(\lambda x.\, t_1)\, t_2] \xrightarrow{\mathbb{E}} E[t_1[x := t_2]]$

An Example Operational Semantics

Example: functions with call-by-name evaluation to weak head normal form.

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Evaluation contexts: $E \in \mathsf{EvalContext} ::= \square \mid (E\, t)$

Evaluation rewriting: $E[(\lambda x.\, t_1)\, t_2] \xrightarrow{\mathbb{E}} E[t_1[x := t_2]]$

Operational meaning:
$$\mathsf{result}(t) = \begin{cases} \mathsf{diverges} & \text{if } \neg\text{has-nf}(\mathbb{E}, t) \\ \mathsf{halt} & \text{if } t \xrightarrow{\mathbb{E},\mathsf{nf}} \lambda x.\, t' \\ \mathsf{stuck} & \text{if } t \xrightarrow{\mathbb{E},\mathsf{nf}} t' \neq \lambda x.\, t'' \end{cases}$$

Rewriting for Program Equivalences

Suppose we want to use the evaluation rewrite rule in arbitrary contexts C, i.e., the usual β rule:

$$C[(\lambda x.\, t_1)\, t_2] \longrightarrow C[t_1[x := t_2]]$$

Rewriting for Program Equivalences

Suppose we want to use the evaluation rewrite rule in arbitrary contexts C, i.e., the usual β rule:

$$C[(\lambda x.\, t_1)\, t_2] \longrightarrow C[t_1[x := t_2]]$$

Example rewrite steps in non-evaluation position:

$$(\lambda y.\, (\lambda w.\, w)\, y) \qquad \longrightarrow (\lambda y.\, y)$$

Rewriting for Program Equivalences

Suppose we want to use the evaluation rewrite rule in arbitrary contexts C, i.e., the usual β rule:

$$C[(\lambda x.\, t_1)\, t_2] \longrightarrow C[t_1[x := t_2]]$$

Example rewrite steps in non-evaluation position:

$$(\lambda y.\, (\lambda w.\, w)\, y) \longrightarrow (\lambda y.\, y)$$

$$(\lambda x.\, xx)\, (\lambda y.\, (\lambda w.\, w)\, y) \longrightarrow (\lambda x.\, xx)\, (\lambda y.\, y)$$

Rewriting for Program Equivalences

Suppose we want to use the evaluation rewrite rule in arbitrary contexts C, i.e., the usual β rule:

$$C[(\lambda x.\, t_1)\, t_2] \longrightarrow C[t_1[x := t_2]]$$

Example rewrite steps in non-evaluation position:

$$(\lambda y.\, (\lambda w.\, w)\, y) \longrightarrow (\lambda y.\, y)$$

$$(\lambda x.\, xx)\, (\lambda y.\, (\lambda w.\, w)\, y) \longrightarrow (\lambda x.\, xx)\, (\lambda y.\, y)$$

Is this *meaning-preserving*, i.e., does $t_1 \longrightarrow t_2$ imply that $\text{result}(t_1) = \text{result}(t_2)$?

Rewriting for Program Equivalences

Suppose we want to use the evaluation rewrite rule in arbitrary contexts C, i.e., the usual β rule:

$$C[(\lambda x.\, t_1)\, t_2] \longrightarrow C[t_1[x := t_2]]$$

Example rewrite steps in non-evaluation position:

$$(\lambda y.\, (\lambda w.\, w)\, y) \longrightarrow (\lambda y.\, y)$$

$$(\lambda x.\, xx)\, (\lambda y.\, (\lambda w.\, w)\, y) \longrightarrow (\lambda x.\, xx)\, (\lambda y.\, y)$$

Is this *meaning-preserving*, i.e., does $t_1 \longrightarrow t_2$ imply that $\text{result}(t_1) = \text{result}(t_2)$?

Is it an *observational equivalence*, i.e., does $t_1 \longrightarrow t_2$ imply $\text{result}(C[t_1]) = \text{result}(C[t_2])$ for any context C?

Overview

- Motivation.
- **The AES framework.**
- Methods for proving meaning preservation.
- Discussion.

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$ set of *terms*

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$	set of *terms*
$\mathbb{S},$	set of *rewrite steps*

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$	set of *terms*
$\mathbb{S},$	set of *rewrite steps*
$\mathbb{R},$	set of *evaluation results*

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$	set of *terms*
$\mathbb{S},$	set of *rewrite steps*
$\mathbb{R},$	set of *evaluation results*
endpoints,	maps $\mathbb{S}$ to $\mathbb{T} \times \mathbb{T}$

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$	set of *terms*
$\mathbb{S},$	set of *rewrite steps*
$\mathbb{R},$	set of *evaluation results*
$\mathsf{endpoints},$	maps $\mathbb{S}$ to $\mathbb{T} \times \mathbb{T}$
$\mathbb{E},$	a subset of $\mathbb{S}$, the *evaluation steps*

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$	set of *terms*
$\mathbb{S},$	set of *rewrite steps*
$\mathbb{R},$	set of *evaluation results*
$\mathsf{endpoints},$	maps $\mathbb{S}$ to $\mathbb{T} \times \mathbb{T}$
$\mathbb{E},$	a subset of $\mathbb{S}$, the *evaluation steps*
$\mathsf{result})$	maps $\mathbb{T}$ to $\mathbb{R}$

Abstract Evaluation Systems

To discuss the issues, the notion of *abstract evaluation system* (AES) will be used.

An AES is a 6-tuple:

$(\mathbb{T},$	set of *terms*
$\mathbb{S},$	set of *rewrite steps*
$\mathbb{R},$	set of *evaluation results*
endpoints,	maps $\mathbb{S}$ to $\mathbb{T} \times \mathbb{T}$
$\mathbb{E},$	a subset of $\mathbb{S}$, the *evaluation steps*
result$)$	maps $\mathbb{T}$ to $\mathbb{R}$

Let the *non-evaluation steps* be $\mathbb{N} = \mathbb{S} \setminus \mathbb{E}$.

Comments on AES Framework (1)

- There is a separate set $\mathbb{S}$ of steps which are not just term pairs.

Comments on AES Framework (1)

- There is a separate set $\mathbb{S}$ of steps which are not just term pairs.
  - This helps distinguish between different redexes that reach the same term, e.g.:

$$((\lambda x.\, xx)\,(\lambda x.\, xx))\,((\lambda x.\, xx)\,(\lambda x.\, xx))$$

Comments on AES Framework (1)

- There is a separate set $\mathbb{S}$ of steps which are not just term pairs.
  - This helps distinguish between different redexes that reach the same term, e.g.:

$$((\lambda x.\, xx)\,(\lambda x.\, xx))\,((\lambda x.\, xx)\,(\lambda x.\, xx))$$

  - So an AES definer does not need to reason about all other redexes in the same term when deciding whether a step is in $\mathbb{E}$ or $\mathbb{N}$.

Comments on AES Framework (1)

- There is a separate set $\mathbb{S}$ of steps which are not just term pairs.
  - This helps distinguish between different redexes that reach the same term, e.g.:

$$((\lambda x.\, xx)\, (\lambda x.\, xx))\, ((\lambda x.\, xx)\, (\lambda x.\, xx))$$

  - So an AES definer does not need to reason about all other redexes in the same term when deciding whether a step is in $\mathbb{E}$ or $\mathbb{N}$.
- Rewriting notation for a subset $\mathcal{S} \subset \mathbb{S}$:

$$t_1 \xrightarrow{\mathcal{S}} t_2 \Leftrightarrow \exists s \in \mathcal{S}.\ \mathsf{endpoints}(s) = (t_1, t_2)$$

Comments on AES Framework (1)

- There is a separate set $\mathbb{S}$ of steps which are not just term pairs.
  - This helps distinguish between different redexes that reach the same term, e.g.:

$$((\lambda x.\, xx)\,(\lambda x.\, xx))\;((\lambda x.\, xx)\,(\lambda x.\, xx))$$

  - So an AES definer does not need to reason about all other redexes in the same term when deciding whether a step is in $\mathbb{E}$ or $\mathbb{N}$.
- Rewriting notation for a subset $\mathcal{S} \subset \mathbb{S}$:

$$t_1 \xrightarrow{\mathcal{S}} t_2 \Leftrightarrow \; \exists s \in \mathcal{S}.\, \mathsf{endpoints}(s) = (t_1, t_2)$$
$$t_1 \xrightarrow{\mathcal{S}_1, \mathcal{S}_2} t_2 \Leftrightarrow \; t_1 \xrightarrow{\mathcal{S}_1 \cap \mathcal{S}_2} t_2$$

Comments on AES Framework (2)

- Evaluation rewriting (i.e., $\xrightarrow{\mathbb{E}}$) must be *subcommutative*. Often, it will be deterministic, but the extra flexibility is there for when needed.

Comments on AES Framework (2)

- Evaluation rewriting (i.e., $\xrightarrow{\mathbb{E}}$) must be *subcommutative*. Often, it will be deterministic, but the extra flexibility is there for when needed.
- The special result value diverges is reserved for terms with non-halting evaluation.

Comments on AES Framework (2)

- Evaluation rewriting (i.e., $\xrightarrow{\mathbb{E}}$) must be *subcommutative*. Often, it will be deterministic, but the extra flexibility is there for when needed.
- The special result value diverges is reserved for terms with non-halting evaluation.
- Evaluation steps must preserve results, i.e., $t_1 \xrightarrow{\mathbb{E}} t_2$ must imply that $\mathsf{result}(t_1) = \mathsf{result}(t_2)$.

Comments on AES Framework (2)

- Evaluation rewriting (i.e., $\xrightarrow{\mathbb{E}}$) must be *subcommutative*. Often, it will be deterministic, but the extra flexibility is there for when needed.
- The special result value $\mathsf{diverges}$ is reserved for terms with non-halting evaluation.
- Evaluation steps must preserve results, i.e., $t_1 \xrightarrow{\mathbb{E}} t_2$ must imply that $\mathsf{result}(t_1) = \mathsf{result}(t_2)$.
- The intention is to model execution where the only way to observe a result is to do evaluation steps as long as possible and then inspect the halted term, which is unique even when evaluation is non-deterministic (a deliberate AES framework limitation).

Example AES

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Example AES

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Contexts: $C \in \mathsf{Context} ::= \Box \mid x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Rewrite steps: $s \in \mathbb{S} ::= (C, ((\lambda x.\, t_1)\, t_2))$

Example AES

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Contexts: $C \in \mathsf{Context} ::= \Box \mid x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Rewrite steps: $s \in \mathbb{S} ::= (C, ((\lambda x.\, t_1)\, t_2))$

Evaluation results: $r \in \mathbb{R} = \{\mathsf{diverges}, \mathsf{stuck}, \mathsf{halt}\}$

Example AES

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Contexts: $C \in \mathsf{Context} ::= \Box \mid x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Rewrite steps: $s \in \mathbb{S} ::= (C, ((\lambda x.\, t_1)\, t_2))$

Evaluation results: $r \in \mathbb{R} = \{\mathsf{diverges}, \mathsf{stuck}, \mathsf{halt}\}$

Rewrite step endpoints:

$$\mathsf{endpoints}(C, (\lambda x.\, t_1)\, t_2) = (C[(\lambda x.\, t_1)\, t_2], C[t_1[x := t_2]])$$

Example AES

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Contexts: $C \in \text{Context} ::= \Box \mid x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Rewrite steps: $s \in \mathbb{S} ::= (C, ((\lambda x.\, t_1)\, t_2))$

Evaluation results: $r \in \mathbb{R} = \{\text{diverges}, \text{stuck}, \text{halt}\}$

Rewrite step endpoints: $\text{endpoints}(C, (\lambda x.\, t_1)\, t_2)$
$= (C[(\lambda x.\, t_1)\, t_2], C[t_1[x := t_2]])$

Evaluation contexts: $E \in \text{EvalContext} ::= \Box \mid (E\, t)$

Evaluation steps: $s \in \mathbb{E} ::= (E, ((\lambda x.\, t_1)\, t_2))$

Useful
Logics,
Types,
Rewriting, and their
Automation

Example AES

Terms: $t \in \mathbb{T} ::= x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Contexts: $C \in \text{Context} ::= \square \mid x \mid (\lambda x.\, t) \mid (t_1\, t_2)$

Rewrite steps: $s \in \mathbb{S} ::= (C, ((\lambda x.\, t_1)\, t_2))$

Evaluation results: $r \in \mathbb{R} = \{\text{diverges}, \text{stuck}, \text{halt}\}$

Rewrite step endpoints: $\text{endpoints}(C, (\lambda x.\, t_1)\, t_2)$

$$= (C[(\lambda x.\, t_1)\, t_2], C[t_1[x := t_2]])$$

Evaluation contexts: $E \in \text{EvalContext} ::= \square \mid (E\, t)$

Evaluation steps: $s \in \mathbb{E} ::= (E, ((\lambda x.\, t_1)\, t_2))$

Operational meaning:

$$\text{result}(t) = \begin{cases} \text{diverges} & \text{if } \neg\text{has-nf}(\mathbb{E}, t) \\ \text{halt} & \text{if } t \xrightarrow{\mathbb{E},\text{nf}} \lambda x.\, t' \\ \text{stuck} & \text{if } t \xrightarrow{\mathbb{E},\text{nf}} t' \neq \lambda x. \end{cases}$$

Overview

- Motivation.
- The AES framework.
- **Methods for proving meaning preservation.**
  - **Previous high-level proof methods.**
  - New high-level proof methods.
  - Low-level proof methods with elementary diagrams.
  - Marks (e.g., finite developments).
- Discussion.

Proving Program Equivalences

- Denotational methods (semantic models).

Proving Program Equivalences

- Denotational methods (semantic models).
- Logical relations. Requires a type system, so hard to use for the full untyped calculus. (An intersection type system can sometimes cover an entire untyped system, but this is difficult.)

Proving Program Equivalences

- Denotational methods (semantic models).
- Logical relations. Requires a type system, so hard to use for the full untyped calculus. (An intersection type system can sometimes cover an entire untyped system, but this is difficult.)
- Operational techniques. Applicative bisimulation and co-induction. Howe's method.

Proving Program Equivalences

- Denotational methods (semantic models).
- Logical relations. Requires a type system, so hard to use for the full untyped calculus. (An intersection type system can sometimes cover an entire untyped system, but this is difficult.)
- Operational techniques. Applicative bisimulation and co-induction. Howe's method.
- This talk will focus on rewriting-based methods: Plotkin [1975], Machkasova and Turbak [2000], Odersky [1993], Ariola and Blom [2002].

Plotkin's Method

Suppose $t_1 \longleftrightarrow t_2$.

Plotkin's Method

Suppose $t_1 \longleftrightarrow t_2$.

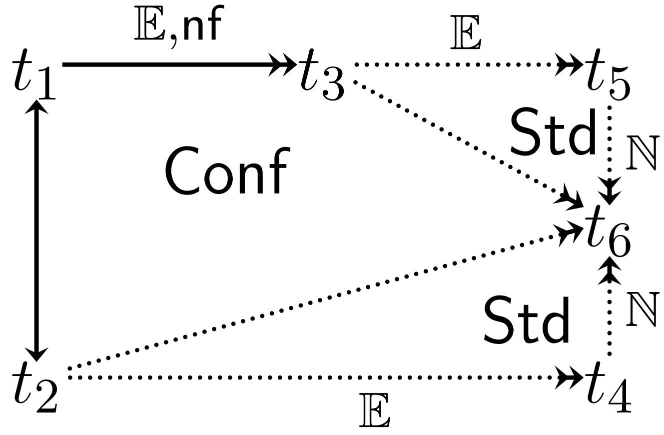
If both diverge, they are assigned same meaning (important!).

Plotkin's Method

Suppose $t_1 \longleftrightarrow t_2$.

If both diverge, they are assigned same meaning (important!).

Suppose evaluation of one halts, maybe t_1. Then:



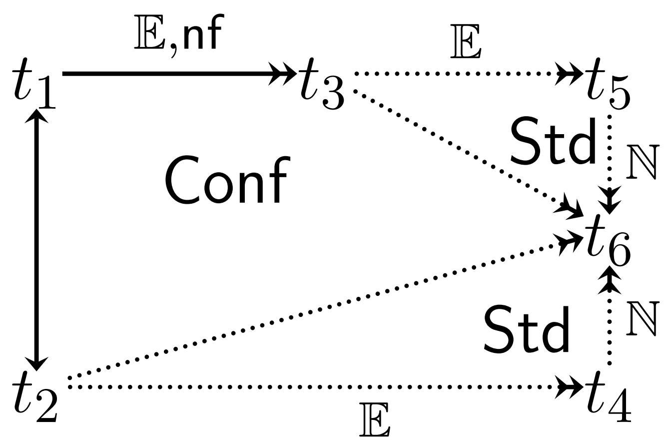
By definition, $\mathsf{result}(t_1) = \mathsf{result}(t_5)$ and $\mathsf{result}(t_2) = \mathsf{result}(t_4)$.

Plotkin's Method

Suppose $t_1 \longleftrightarrow t_2$.

If both diverge, they are assigned same meaning (important!).

Suppose evaluation of one halts, maybe t_1. Then:



By definition, $\mathsf{result}(t_1) = \mathsf{result}(t_5)$ and $\mathsf{result}(t_2) = \mathsf{result}(t_4)$.

Because t_5 has halted and $\mathbb{N}$-conversion preserves both this fact and results (important!), $\mathsf{result}(t_5) = \mathsf{result}(t_4)$.

Comments on Plotkin's Method

- Plotkin [1975] originally developed it for the call-by-name and call-by-value λ-calculus.

Comments on Plotkin's Method

- Plotkin [1975] originally developed it for the call-by-name and call-by-value λ-calculus.
- Other researchers have used it for variations on the λ-calculus, e.g., λ-calculus plus assignments and continuations [Felleisen and Hieb, 1992] and the call-by-need λ-calculus [Ariola and Felleisen, 1997; Maraist, Odersky, and Wadler, 1998].

Comments on Plotkin's Method

- Plotkin [1975] originally developed it for the call-by-name and call-by-value λ-calculus.
- Other researchers have used it for variations on the λ-calculus, e.g., λ-calculus plus assignments and continuations [Felleisen and Hieb, 1992] and the call-by-need λ-calculus [Ariola and Felleisen, 1997; Maraist, Odersky, and Wadler, 1998].
- Requiring confluence prevents using some approaches for reasoning about mutually recursive bindings [Ariola and Klop, 1997].

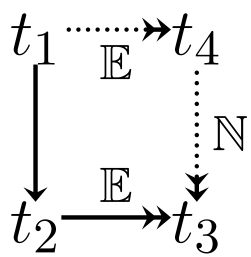
Comments on Plotkin's Method

- Plotkin [1975] originally developed it for the call-by-name and call-by-value λ-calculus.
- Other researchers have used it for variations on the λ-calculus, e.g., λ-calculus plus assignments and continuations [Felleisen and Hieb, 1992] and the call-by-need λ-calculus [Ariola and Felleisen, 1997; Maraist, Odersky, and Wadler, 1998].
- Requiring confluence prevents using some approaches for reasoning about mutually recursive bindings [Ariola and Klop, 1997].
- Requiring standardization tends to force the evaluation contexts and rewrite rules to look arbitrarily deep into the term and inspect an arbitrary number of tree nodes.

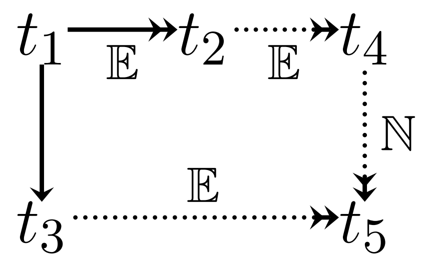
Lift & Project Method

Machkasova and Turbak [2000] introduced the *Lift* and *Project* diagrams:

Lift :



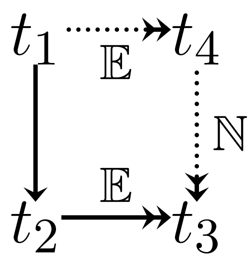
Proj :



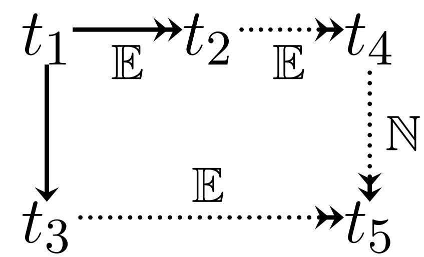
Lift & Project Method

Machkasova and Turbak [2000] introduced the *Lift* and *Project* diagrams:

Lift :



Proj :

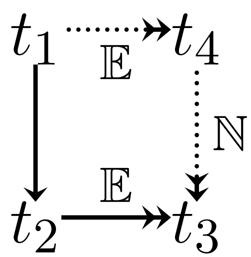


Lift and Project can be substituted for confluence and standardization when proving meaning preservation.

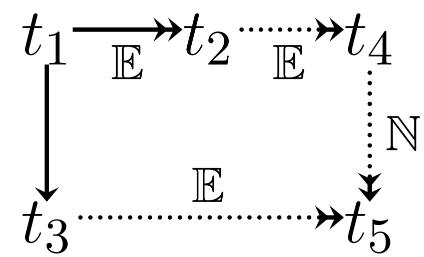
Lift & Project Method

Machkasova and Turbak [2000] introduced the *Lift* and *Project* diagrams:

Lift :



Proj :



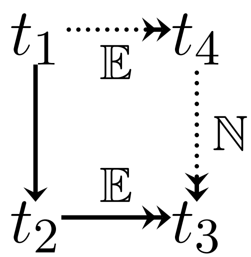
Lift and Project can be substituted for confluence and standardization when proving meaning preservation.

The key benefit is that Lift and Project do not imply confluence, although Lift is equivalent to standardization.

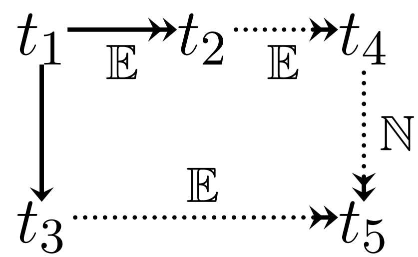
Lift & Project Method

Machkasova and Turbak [2000] introduced the *Lift* and *Project* diagrams:

Lift :



Proj :



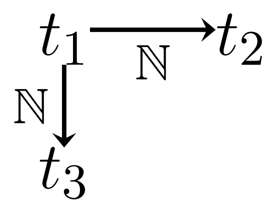
Lift and Project can be substituted for confluence and standardization when proving meaning preservation.

The key benefit is that Lift and Project do not imply confluence, although Lift is equivalent to standardization.

In particular, Lift and Project can be used to prove correctness of Ariola/Blom/Klop-style rewrite rules for letrec.

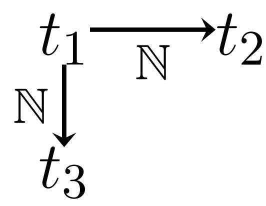
Comparison of Previous Proof Methods

- Lift & Project can handle cases for which confluence & standardization fail, e.g.:

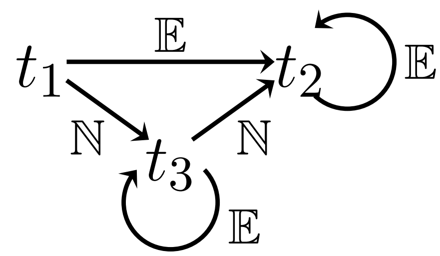


Comparison of Previous Proof Methods

- Lift & Project can handle cases for which confluence & standardization fail, e.g.:



- (*New result:*) Confluence & standardization can handle cases for which Lift & Project fail, e.g.:



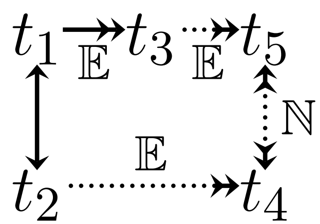
Overview

- Motivation.
- The AES framework.
- **Methods for proving meaning preservation.**
  - Previous high-level proof methods.
  - **New high-level proof methods.**
  - Low-level proof methods with elementary diagrams.
  - Marks (e.g., finite developments).
- Discussion.

Weakening the Proof Burden (1)

- By carefully inspecting how confluence & standardization and Lift & Project prove meaning preservation, we obtained the following weaker *Lift/Project* (LP) diagram which implies meaning preservation:

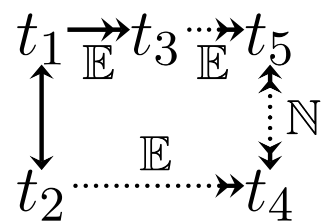
LP :



Weakening the Proof Burden (1)

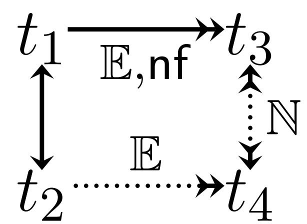
- By carefully inspecting how confluence & standardization and Lift & Project prove meaning preservation, we obtained the following weaker *Lift/Project* (LP) diagram which implies meaning preservation:

LP :



- We further weakened LP to the following *Lift/Project when Terminating* (LPT) diagram:

LPT :



Weakening the Proof Burden (2)

- The LPT diagram gives the fundamental essence of this family of proof methods and is the weakest condition we know of its kind. It is weaker than similar conditions in the related work by Ariola and Blom [2002].

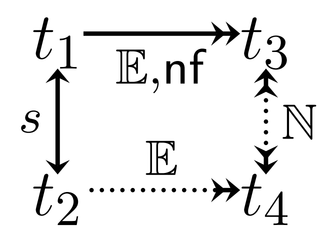
Weakening the Proof Burden (2)

- The LPT diagram gives the fundamental essence of this family of proof methods and is the weakest condition we know of its kind. It is weaker than similar conditions in the related work by Ariola and Blom [2002].
- We further weaken the conditions by parameterizing all diagrams on rewrite steps or step sets.

Weakening the Proof Burden (2)

- The LPT diagram gives the fundamental essence of this family of proof methods and is the weakest condition we know of its kind. It is weaker than similar conditions in the related work by Ariola and Blom [2002].
- We further weaken the conditions by parameterizing all diagrams on rewrite steps or step sets.
  - E.g., actual definition of LPT is:

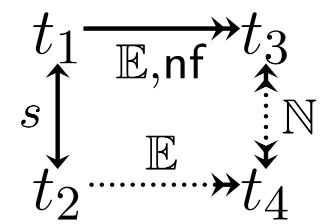
$$s \in \mathsf{LPT} \iff$$



Weakening the Proof Burden (2)

- The LPT diagram gives the fundamental essence of this family of proof methods and is the weakest condition we know of its kind. It is weaker than similar conditions in the related work by Ariola and Blom [2002].
- We further weaken the conditions by parameterizing all diagrams on rewrite steps or step sets.
  - E.g., actual definition of LPT is:

$s \in \mathsf{LPT} \;\Leftrightarrow$



  - This is important because sometimes different methods are needed for different parts of a rewriting system.

Overview

- Motivation.
- The AES framework.
- **Methods for proving meaning preservation.**
  - Previous high-level proof methods.
  - New high-level proof methods.
  - **Low-level proof methods with elementary diagrams.**
  - Marks (e.g., finite developments).
- Discussion.

Proving the High-Level Diagrams

- Diagrams like confluence, standardization, Lift, Proj, LP, and LPT can be used to prove meaning preservation, but they are themselves quite hard to prove, because the diagrams are quite high-level and abstract.

Proving the High-Level Diagrams

- Diagrams like confluence, standardization, Lift, Proj, LP, and LPT can be used to prove meaning preservation, but they are themselves quite hard to prove, because the diagrams are quite high-level and abstract.
- We present 2 new meaning preservation proof methods which are *low-level* because their conditions are only *elementary diagrams* and simple (to understand, not necessarily to prove) kinds of termination.

Proving the High-Level Diagrams

- Diagrams like confluence, standardization, Lift, Proj, LP, and LPT can be used to prove meaning preservation, but they are themselves quite hard to prove, because the diagrams are quite high-level and abstract.
- We present 2 new meaning preservation proof methods which are *low-level* because their conditions are only *elementary diagrams* and simple (to understand, not necessarily to prove) kinds of termination.
- It is expected that a rewriting system will be partitioned into step sets that are closed under "residuals w.r.t. evaluation" and the right method will be used for each partition. Often, each partition will contain all of the steps for some subset of the rewrite rules.

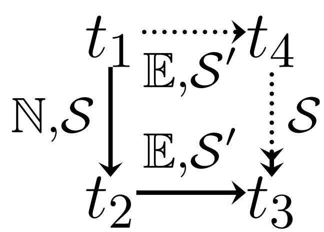
Low-level Method 1

Well Behaved with Standardization:

$$\mathsf{WB{+}Std}(\mathcal{S}) \iff \mathsf{Trm}(\mathbb{E} \cap \mathcal{S}) \wedge \mathsf{WL1}(\mathcal{S}, \mathcal{S}) \wedge \mathsf{WL1}(\mathcal{S}, \mathbb{S}) \wedge \mathsf{WP1}($$

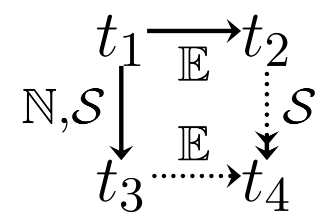
Weak Lift 1-Step:

$$\mathsf{WL1}(\mathcal{S}, \mathcal{S}') \iff$$



Weak Project 1-Step:

$$\mathsf{WP1}(\mathcal{S}) \iff$$



Useful for difficult rewrite step sets which do not have finite developments, e.g., Ariola/Blom/Klop-style letrec rewrite rules.

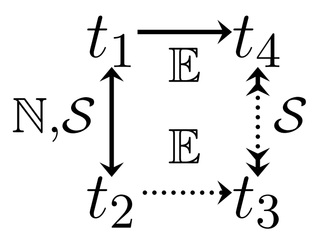
Low-level Method 2

Well Behaved without Standardization:

$$\mathsf{WB“Std}(\mathcal{S}) \iff \mathsf{Trm}(\mathcal{S}) \wedge \mathsf{LConf}(\mathcal{S}) \wedge \mathsf{WLP1}(\mathcal{S}) \wedge \mathsf{NE}(\mathcal{S})$$

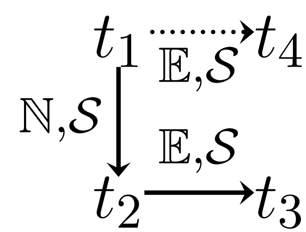
Weak Lift/Project 1-Step:

$$\mathsf{WLP1}(\mathcal{S}) \iff$$



$\mathbb{N}$-Steps Do Not Create $\mathbb{E}$-Step

$$\mathsf{NE}(\mathcal{S}) \iff$$



Useful for difficult rewrite step sets which do not have standardization but do have termination.

Overview

- Motivation.
- The AES framework.
- **Methods for proving meaning preservation.**
  - Previous high-level proof methods.
  - New high-level proof methods.
  - Low-level proof methods with elementary diagrams.
  - **Marks (e.g., finite developments).**
- Discussion.

When Termination Properties Fail

- Sometimes, a desired termination property fails for a rewrite step set $\mathcal{S}$ generated by some rewrite rule(s), but holds for $\mathcal{S} \cap \mathbb{M}$ where $\mathbb{M}$ is a set of *marked* steps.

When Termination Properties Fail

- Sometimes, a desired termination property fails for a rewrite step set $\mathcal{S}$ generated by some rewrite rule(s), but holds for $\mathcal{S} \cap \mathbb{M}$ where $\mathbb{M}$ is a set of *marked* steps.
- The marks typically force termination by forbidding contracting unmarked redexes and ensuring “created” redexes are unmarked.

When Termination Properties Fail

- Sometimes, a desired termination property fails for a rewrite step set $\mathcal{S}$ generated by some rewrite rule(s), but holds for $\mathcal{S} \cap \mathbb{M}$ where $\mathbb{M}$ is a set of *marked* steps.
- The marks typically force termination by forbidding contracting unmarked redexes and ensuring "created" redexes are unmarked.
- The rewriting system is embedded in a larger marked system with additional marked terms and rewrite steps. Proving the larger system correct is enough.

When Termination Properties Fail

- Sometimes, a desired termination property fails for a rewrite step set $\mathcal{S}$ generated by some rewrite rule(s), but holds for $\mathcal{S} \cap \mathbb{M}$ where $\mathbb{M}$ is a set of *marked* steps.
- The marks typically force termination by forbidding contracting unmarked redexes and ensuring "created" redexes are unmarked.
- The rewriting system is embedded in a larger marked system with additional marked terms and rewrite steps. Proving the larger system correct is enough.
- We give conditions on marking such that proving LPT for $\mathcal{S} \cap \mathbb{M}$ (i.e., the marked fragment of the larger marked system) is sufficient to prove LPT for $\mathcal{S}$.

Overview

- Motivation.
- The AES framework.
- Methods for proving meaning preservation.
- **Discussion.**

Related Work

- Our work is a direct successor to the work of Machkasova and Turbak [2000].

Related Work

- Our work is a direct successor to the work of Machkasova and Turbak [2000].
- The work of Ariola and Blom [2002] has important similarities at a deep level. Their framework does not make it easy to prove operational properties, e.g., the user must prove a connection between *infinite normal forms* and operational behavior. Also, there are no low-level abstract proof methods.

Related Work

- Our work is a direct successor to the work of Machkasova and Turbak [2000].
- The work of Ariola and Blom [2002] has important similarities at a deep level. Their framework does not make it easy to prove operational properties, e.g., the user must prove a connection between *infinite normal forms* and operational behavior. Also, there are no low-level abstract proof methods.
- Odersky [1993] gives conditions that a transformation is an observational equivalence. Despite similarities, the formal presentation is quite different and tied to a particular syntactic formalism.

Conclusions

- Overall, the meaning-preservation proof methods we present gather together the strengths of existing methods and improve on them in a number of ways.

Conclusions

- Overall, the meaning-preservation proof methods we present gather together the strengths of existing methods and improve on them in a number of ways.
- Our proof methods are designed to be easy for someone who is not a rewriting specialist to read, understand, and apply to their programming language calculi.

Conclusions

- Overall, the meaning-preservation proof methods we present gather together the strengths of existing methods and improve on them in a number of ways.
- Our proof methods are designed to be easy for someone who is not a rewriting specialist to read, understand, and apply to their programming language calculi.
- We expect that our methods will help in making the expertise of the rewriting community accessible and useful to the outside world.

References

Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Ann. Pure Appl. Logic*, 117(1–3): 95–168, 2002.

Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 3(7), May 1997.

Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inform. & Comput.*, 139:154–233, 1997.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.*, 102:235–271, 1992.

Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In *Programming Languages & Systems, 9th European Symp. Programming*, volume 1782 of *LNCS*, pages 260–274. Springer-Verlag, 2000. ISBN 3-540-67262-1. URL `http://www.church-project.org/reports/ele`

John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8(3), May 1998.

Martin Odersky. A syntactic method for proving observational equivalences. Research Report YALEU/DCS/RR-964, Yale Univ., Dept. of Comp. Science, May 1993.

Gordon D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.