Reasoning with Actions

S. B. Lassen
BRICS*, University of Aarhus
email: thales@brics.aau.dk
URL: http://www.daimi.aau.dk/~thales

Abstract Action semantics is a semantic description framework with very good pragmatic properties but a rather weak theory for reasoning about programs. A strong action theory would be of great practical use, however. It would make it possible to reason about the large class of programming languages that can be described in action semantics.

This paper develops the foundations for a richer action theory, by bringing together concepts and techniques from testing theory for processes and from work on operational reasoning about functional programs. Semantic preorders and equivalences in the action semantics setting are studied and a useful operational technique for establishing testing equivalences is presented.

1 Introduction

In this paper we develop a richer theory for reasoning about programs in *action semantics* (AS). Because AS is a general semantic description framework, our work has a great practical scope. A strong action theory would offer techniques for reasoning about programs in any programming language that can be described in AS.

1.1 Program reasoning in action semantics

AS is a denotational-style semantics. An *action semantic description* (ASD) maps program terms to the semantic entities they denote. The semantic entities are called *actions*.

Normally, we shall regard an ASD of a programming language as the formal definition of the language it describes, hence it is trivially adequate. Because ASDs are also compositional, language terms are operationally equivalent if their

*Basic Research in Computer Science, Centre of the Danish National Research Foundation.

denotations, i.e. actions, are equivalent—provided this action equivalence is defined properly.

Therefore, with the right notion of action equivalence, equivalences of programs can be established by showing their actions equivalent. This motivates the undertaking of this paper. In the rest of the paper, we shall examine notions of semantic equivalence of actions and techniques for reasoning about these action equivalences.

1.2 Existing action theory

Current action theory mainly consists of a series of action laws that codify algebraic properties of actions. In [16, App.C] the meaning of actions is defined in a structural operational semantics, a testing equivalence of actions is defined (a sort of MAY equivalence [2]), and a bisimulation technique for establishing equivalences is developed. By means of bisimulation, all action laws can be shown to respect testing equivalence.

Practical applications are few. Action laws are applied in the ACTRESS compiler generator for optimisations [21] but it remains doubtful whether the applicability of present techniques goes beyond simple, local transformations. Especially for reasoning about memory operations neither laws nor bisimulation seem to have much to offer.

1.3 Background for our work

Actions have functional, imperative, as well as concurrent features. There is thus a great range of places to look for inspiration for developing action theory.

Testing and bisimulation are notions from process calculus theory. Following up on the former, we will examine the full range of standard testing preorders and equivalences for actions. But our main source of inspiration comes from work on operational reasoning about impure functional languages.

Ian Mason and Carolyn Talcott have developed powerful operational techniques for deciding program equivalences that involve memory [13, 11]. A drawback of their approach is that it deals with a particular functional language. They have considered an extension to a concurrent ACTOR language [1]. Nevertheless, it is not clear how their techniques apply to other programming languages.

In contrast, if we can develop a strong action theory, it will apply readily to the large class of programming languages that admit an ASD. We shall in the following try to fit actions into Mason and Talcott's framework and generalise their techniques to nondeterminism as it appears in actions.

1.4 Overview

Section 2 introduces AS as well as the actions and action laws to be treated in the rest of the paper. In order to fit actions into Mason and Talcott's framework, Section 3 defines a so-called reduction semantics for action. In Section 4, testing preorders and equivalences are defined and a context lemma is established. Section 5 develops a technique for proving actions test preordered, based on the context lemma, and sketches the resulting inequational and equational action theory. A discussion of future and related work concludes the paper.

2 Action Semantics

Action semantics (AS) is a semantic description framework with very good pragmatic properties [16]. It has been used to formalise a wide range of realistic programming languages, as demonstrated by complete descriptions of Pascal [19] and ANDF [9] as well as of numerous other functional, procedural, and object-oriented languages. Moreover, a number of compiler generators have been based on AS [23, 3, 21, 22].

2.1 Action semantic descriptions

An action semantic description (ASD) consists of compositional semantic functions that map abstract syntax to the universal domain of actions. Suppose we have an imperative language with a syntactic category Statement in the abstract syntax. We can have a semantic function such as:

- execute _ :: Statement → action .

Some example semantic equations:

(1) execute "**skip**" = complete .

(2) execute ⟦ S_1:Statement ";" S_2:Statement ⟧ =
    execute S_1 and then execute S_2 .

(3) execute ⟦ "**loop**" S:Statement ⟧ =
    (unfolding
      (execute S and then unfold))
    trap complete .

(4) execute "**exit**" = escape .

The actions appearing on the right hand sides of the semantic equations will be explained shortly.

2.2 Action notation

Action notation (AN) is the specification language of AS for expressing actions. It is a rich language which expresses common computational notions in a direct way. AN is structured into various *facets* according to the information being processed:

- The **basic** facet is concerned with *control*, no information is processed. Higher-order, exceptional, nondeterministic, and interleaving control flow are expressible.
- The **functional** facet—processes transient data; features unbounded choice.
- The **declarative** facet—deals with bindings; control, data, and binding flow may be combined in various ways.
- The **imperative** facet—operates on a single-threaded storage.
- The **communicative** facet—connects distributed agents which communicate asynchronously.

The semantic equations above employ only the basic facet of AN. For simplicity we shall focus on the basic actions in this paper.

2.3 Basic actions

AN is a combinator based notation. Actions are either *primitives* or compounds built using *combinators*. The outcome of performing an action is

completed, normal termination (subsequent processing continues normally);

escaped, exceptional termination (transfers control to an exception handler);

failed, abnormal termination (aborts current processing); or

divergent, no termination.

The basic primitives are:

complete, escape, and fail, completes, fails, and escapes, respectively.

check _, takes a truth value argument and completes if it is true and fails if it is false.

enact _, invokes the abstraction yielded by its argument—an abstraction is an abstracted (or 'reified') action.

unfold, marks a point of recursion in the body of the unary combinator unfolding (see below).

The basic combinators are:

and then, binary sequential combinator.

and, binary combinator; performs its two subactions in any, arbitrarily interleaved order.

trap, binary combinator; performs its second subaction as an exception handler if the first subaction escapes.

or, binary choice combinator; if one subaction fails, the other is chosen, if neither subaction fails, one is chosen nondeterministically.

unfolding, unary recursion combinator; the primitive unfold in the subaction of unfolding denotes a recursive unwinding of the subaction. A divergent action is most shortly written diverge = unfolding unfold .

A grammar of basic AN can be specified in unified algebras [18] as follows:

- action-primitive = complete | escape | fail |
  check yielder | enact yielder | unfold .
- action = action-primitive | action and then action | action and action |
  action trap action | action or action | unfolding action .
- yielder = true | false | abstraction of action .

The arguments to the parameterised primitives check and enact are called yielders. In the basic facet, yielders are just constants, either truth values or abstractions.

To illustrate how actions give semantics to programming language terms, consider two statements from the example language from before:

(i) **skip** .
(ii) **loop exit** .

The semantic function execute maps them to a primitive and to a compound action, respectively:

(i) complete .
(ii) | unfolding
| | escape and then unfold
trap complete .

The latter escapes inside the body of unfolding, is then caught by trap, and the whole action completes.

Basic actions are not very expressive in themselves. Basic actions can just diverge or terminate in three different ways. In particular, they cannot process data in any way. Only when fused with data processing facets the expressive power of basic actions becomes interesting. Nevertheless, the basic facet suffices to illustrate the main concepts and techniques which we shall present.

2.4 Action laws

There are a series of 'action laws' that characterise actions. The following are a sample of basic action laws:

(1) $\mathsf{check\ true} = \mathsf{complete}$; $\mathsf{check\ false} = \mathsf{fail}$.

(2) $\mathsf{unfolding}\ A = A\ @\ \mathsf{unfolding}\ A$.

(3) $(A_1\ \mathsf{and\ then}\ A_2)\ \mathsf{and\ then}\ A_3 = A_1\ \mathsf{and\ then}\ (A_2\ \mathsf{and\ then}\ A_3)$.

(4) $\mathsf{complete\ and\ then}\ A = A\ \mathsf{and\ then\ complete} = A$.

(5) $\mathsf{escape\ and\ then}\ A = \mathsf{escape}$; $\mathsf{fail\ and\ then}\ A = \mathsf{fail}$.

(6) $(A_1\ \mathsf{trap}\ A_2)\ \mathsf{trap}\ A_3 = A_1\ \mathsf{trap}\ (A_2\ \mathsf{trap}\ A_3)$.

(7) $\mathsf{escape\ trap}\ A = A\ \mathsf{trap\ escape} = A$.

(8) $\mathsf{complete\ trap}\ A = \mathsf{complete}$; $\mathsf{fail\ trap}\ A = \mathsf{fail}$.

Law (1) characterises check. Law (2) states that unfolding is equal to its own unwinding (the operator @ substitutes its second argument for all 'free' occurrences of the primitive unfold in the first argument, $A_1 @ A_2$ = "$A_1[A_2/\mathsf{unfold}]$"). Laws (3)–(5) say that and then is associative, has complete as unit, and is left-absorbed by escape and fail. Laws (6)–(8) are corresponding laws for trap.

As an example application of these laws, the example actions (i)–(ii) above can be shown equal using laws (4), (2), (7).

A catalogue of action laws for all facets of AN may be found in [16, App.B]. In [15], a precursor of AS, such laws served as an algebraic semantics of actions. However, establishing laws sufficient to define full action notation in this way doesn't seem feasible.

3 Reduction semantics

Mason and Talcott operate on the basis of a 'reduction semantics' for their language. A reduction semantics is a small-step operational semantics that represents control and state syntactically.

Our first step towards fitting actions into Mason and Talcott's framework is to define a reduction semantics for basic AN. (A preliminary sketch was presented in [12].) The essential novelty compared to [16, App.C] is that control is represented by means of evaluation contexts.

3.1 Decomposition

An *evaluation context* [6] (or *reduction context* [13]) E is an action with a hole at a legal point of execution in the action. Either just a hole [], a hole to the left of the sequential combinators (*seq* is either of and then , trap), or a hole on either side of the symmetric combinators (*sym* is either of and , or).

- evaluation-context = [ ] | evaluation-context *seq* action
  | evaluation-context *sym* action
  | action *sym* evaluation-context .

A *redex* R is any action that can make a transition directly (see the evaluation rules below). $E[R]$ denotes the action obtained by filling R into the hole in E.

In an action $E[R]$, the evaluation context E corresponds to the point of execution in the program, the program pointer, or current continuation. The redex R is the current instruction.

When an action A can be written as an evaluation context E filled with a redex R, viz. $A = E[R]$, we say that A *decomposes* into E, R. In the following example, there are two possible decompositions of A. So decomposition is not unique.

Example 3.1 *Let* $A =$ check true or (check false and then escape).
Then $A = E_1$[check true] $= E_2$[check false] ,
where $E_1 =$ [ ] or (check false and then escape) ;
$E_2 =$ check true or ([ ] and then escape) .

3.2 Evaluation

The evaluation rules decompose an action and replace the redex with a reduced (or unwound) action.

E : evaluation-context ; A, A_1, A_2 : action ; Y : yielder $\Rightarrow$

$$\text{(1)}\quad E[\mathsf{check}\ Y] \mapsto \begin{cases} E[\mathsf{complete}] & \textit{if}\ \ Y = \mathsf{true} \\ E[\mathsf{fail}] & \textit{if}\ \ Y = \mathsf{false}\ . \end{cases}$$

$$\text{(2)}\quad E[\mathsf{enact}\ Y] \mapsto E[A] \quad \textit{if}\ \ Y = \mathsf{abstraction\ of}\ A\ .$$

$$\text{(3)}\quad E[\mathsf{unfolding}\ A] \mapsto E[A \ @\ \mathsf{unfolding}\ A]\ .$$

$$\text{(4)}\quad E[A_1\ \mathsf{and\ then}\ A_2] \mapsto \begin{cases} E[A_1] & \textit{if}\ \ A_1 : \mathsf{escape}\ |\ \mathsf{fail} \\ E[A_2] & \textit{if}\ \ A_1 = \mathsf{complete}\ . \end{cases}$$

$$\text{(5)}\quad E[A_1\ \mathsf{and}\ A_2] \mapsto \begin{cases} E[A_1] & \textit{if}\ \ A_1 : \mathsf{escape}\ |\ \mathsf{fail}\ \ \textit{or}\ \ A_2 = \mathsf{complete} \\ E[A_2] & \textit{if}\ \ A_1 = \mathsf{complete}\ \ \textit{or}\ \ A_2 : \mathsf{escape}\ |\ \mathsf{fail}\ . \end{cases}$$

$$\text{(6)}\quad E[A_1\ \mathsf{trap}\ A_2] \mapsto \begin{cases} E[A_1] & \textit{if}\ \ A_1 : \mathsf{complete}\ |\ \mathsf{fail} \\ E[A_2] & \textit{if}\ \ A_1 = \mathsf{escape}\ . \end{cases}$$

$$\text{(7)}\quad E[A_1\ \mathsf{or}\ A_2] \mapsto \begin{cases} E[A_1] & \textit{if}\ \ A_1 : \mathsf{complete}\ |\ \mathsf{escape}\ \ \textit{or}\ \ A_2 = \mathsf{fail} \\ E[A_2] & \textit{if}\ \ A_1 = \mathsf{fail}\ \ \mathsf{em\ or}\ \ A_2 : \mathsf{complete}\ |\ \mathsf{escape}\ . \end{cases}$$

Call an action *terminated* if it is any of complete, escape, fail.

The following useful characterisation of evaluation can be shown. If an action makes a transition and we can compose the action as an evaluation context filled with a non-terminated subaction, then either (i) the transition chooses a redex in our subaction, or else a parallel redex is used which either (ii) doesn't affect our subaction, or (iii) eliminates it.

Lemma 3.1 (choice of execution point) *If* $E[A] \mapsto B'$ *and* A *is not terminated, then either* (i) $B' = E[B]$ *where* $A \mapsto B$ *,*
or (ii) $B' = E'[A]$ *and, for all* A', $E[A'] \mapsto E'[A']$ *,*
or (iii) $E[A'] \mapsto B'$ *for all* A' *.*

4 Testing

In this section, we shall define test preorders and equivalences for actions. We also present a tractable characterisation, based on evaluation contexts.

4.1 Definedness

A *computation* from some action A is a sequence of transitions, $A \mapsto A_1 \mapsto A_2 \mapsto \cdots$ We say that a computation is *successful* if it is finite and the final A_n is either complete or escape. Define two 'definedness' predicates for actions. Action A *may* succeed, $A \downarrow_{\text{MAY}}$, if there exists a successful computation from A. A *must* succeed, $A \downarrow_{\text{MUST}}$, if, moreover, all computations from A are successful.

More formally, let success = complete **|** escape, then:

Definition 4.1 (defined) $\downarrow_{\text{MAY}}$ *and* $\downarrow_{\text{MUST}}$ *are the least predicates satisfying*

$$A \downarrow_{\text{MAY}} \quad \textit{if} \quad A \mapsto^* S \ \ \textit{for some} \ \ S{:}\mathsf{success} \ .$$

$$A \downarrow_{\text{MUST}} \quad \textit{if} \quad A \downarrow_{\text{MAY}} \ \ \textit{and} \ \ (\ A \mapsto B \ \ \textit{implies} \ \ B \downarrow_{\text{MUST}} \ ) \ .$$

We will need the following inductive formulation for induction proofs:

$$\begin{array}{lllll}
\downarrow_{\text{MAY}} & = \bigcup_{\lambda<\omega_0} \downarrow^{\lambda}_{\text{MAY}} & \textit{where} & \downarrow^{0}_{\text{MAY}} & = \ \mathsf{success} \ , \\
 & & & \downarrow^{\lambda}_{\text{MAY}} & = \ \{ \ A \ \mid \ \exists \ B \in \bigcup_{\kappa<\lambda} \downarrow^{\kappa}_{\text{MAY}} \ . \ A \mapsto B \ \} \ . \\
\downarrow_{\text{MUST}} & = \bigcup_{\lambda<\omega_0} \downarrow^{\lambda}_{\text{MUST}} & \textit{where} & \downarrow^{0}_{\text{MUST}} & = \ \mathsf{success} \ , \\
 & & & \downarrow^{\lambda}_{\text{MUST}} & = \ \{ \ A \in \downarrow_{\text{MAY}} \ \mid \ \forall \ B \ \textit{s.t.} \ A \mapsto B. \\
 & & & & \qquad\qquad B \in \bigcup_{\kappa<\lambda} \downarrow^{\kappa}_{\text{MUST}} \ \} \ .
\end{array}$$

Basic action notation is finitely branching and by König's lemma we only need to do induction over the finite ordinals (all $\lambda < \omega_0$). If we add the functional facet which introduces countably branching nondeterminism, we must do induction over countable ordinals (all $\lambda < \omega_1$).

4.2 Testing equivalence

Now we can define three testing equivalences. We define them as preorders which induce corresponding equivalences.

Definition 4.2 (test preorders)

$$\begin{array}{rcl} A_1 \mathrel{\sqsubseteq\!\!\!\sim}_{\text{MAY}} A_2 & \textit{iff} & \forall\, C.\ C[A_1]\downarrow_{\text{MAY}} \;\Rightarrow\; C[A_2]\downarrow_{\text{MAY}}\ . \\ A_1 \mathrel{\sqsubseteq\!\!\!\sim}_{\text{MUST}} A_2 & \textit{iff} & \forall\, C.\ C[A_1]\downarrow_{\text{MUST}} \;\Rightarrow\; C[A_2]\downarrow_{\text{MUST}}\ . \\ A_1 \mathrel{\sqsubseteq\!\!\!\sim} A_2 & \textit{iff} & A_1 \mathrel{\sqsubseteq\!\!\!\sim}_{\text{MAY}} A_2 \quad \textit{and} \quad A_1 \mathrel{\sqsubseteq\!\!\!\sim}_{\text{MUST}} A_2\ . \end{array}$$

C ranges over arbitrary action contexts, actions with any number of holes anywhere (also inside abstractions, unfoldings, etc.) These testing preorders generalise conventional operational preorder to the nondeterministic case.

The induced equivalences are:

$$\begin{array}{rcl} \cong_{\text{MUST}} & = & \mathrel{\sqsubseteq\!\!\!\sim}_{\text{MUST}} \cap \mathrel{\sqsupseteq\!\!\!\sim}_{\text{MUST}}\ , \\ \cong_{\text{MAY}} & = & \mathrel{\sqsubseteq\!\!\!\sim}_{\text{MAY}} \cap \mathrel{\sqsupseteq\!\!\!\sim}_{\text{MAY}}\ , \\ \cong & = & \mathrel{\sqsubseteq\!\!\!\sim} \cap \mathrel{\sqsupseteq\!\!\!\sim}\ . \end{array}$$

Call the three notions of preorder/equivalence MAY, MUST, and TEST, respectively.

These equivalences are, by definition, the weakest equivalences such that, in any context, equivalent subterms are interchangeable without any observable effect. One chooses MAY, MUST, or TEST, dependent on one's view of observation: whether one ignores divergence, regards divergence as catastrophic, or demands equal divergence properties, respectively.

The testing equivalence defined for actions in [16, C.4] corresponds to our MAY equivalence. (Both the notion of test and the notion of observation used there are a little less standard.) We shall instead take the TEST equivalence as basic. It is a stronger equivalence, and thus a 'safe compromise' between the MAY and the MUST view of divergence, and it turns out that the action laws hold even for TEST equivalence.

Having all the test preorders, we can express more properties of actions by a variety of inequational laws. This will be exemplified in Section 5.

4.3 Alternative characterisation

The definition of test preorders is difficult to use to prove actions preordered or equivalent. We need an alternative characterisation which gives a tractable way of determining equivalence. This approach is used in test theory [10] and it is central to Mason and Talcott's operational work. Inspired by the latter we have shown the following 'evaluation' preorders to characterise the test preorders:

Definition 4.3 (evaluation preorders)

$$\begin{array}{rcl} A_1 \lesssim_{\text{MAY}} A_2 & \textit{iff} & \forall\, E.\ E[A_1]\downarrow_{\text{MAY}} \;\Rightarrow\; E[A_2]\downarrow_{\text{MAY}}\ . \\ A_1 \lesssim_{\text{MUST}} A_2 & \textit{iff} & \forall\, E.\ E[A_1]\downarrow_{\text{MUST}} \;\Rightarrow\; E[A_2]\downarrow_{\text{MUST}}\ . \end{array}$$

These preorders only quantify over evaluation contexts, not arbitrary contexts. This turns out to be much more tractable. The following lemma is the key to all reasoning about test preorders and equivalences:

Lemma 4.1 (basic context lemma)

$$A_1 \sqsubseteq_{\text{MAY}} A_2 \quad \textit{iff} \quad A_1 \lesssim_{\text{MAY}} A_2 \ .$$
$$A_1 \sqsubseteq_{\text{MUST}} A_2 \quad \textit{iff} \quad A_1 \lesssim_{\text{MUST}} A_2 \ .$$

Proof (outline): Proceed as in the '**ciu**' proof in [11]:

The 'only if's are immediate because evaluation contexts are instances of arbitrary contexts.

The core of the 'if' directions consists of showing the evaluation preorders to be precongruences for basic AN. Then assume $A_1 \lesssim_{\text{MAY}} A_2$ and show $A_1 \sqsubseteq_{\text{MAY}} A_2$ as follows: First of all, $C[A_1] \lesssim_{\text{MAY}} C[A_2]$, for any C, because $\lesssim_{\text{MAY}}$ is a precongruence. Therefore $C[A_1] \downarrow_{\text{MAY}} \ \Rightarrow \ C[A_2] \downarrow_{\text{MAY}}$. Hence $A_1 \sqsubseteq_{\text{MAY}} A_2$. Likewise for MUST. □

The formalisation and proof of the basic context lemma have been carefully phrased such that they generalise to extensions of basic AN. We have investigated additions of functional, declarative, and imperative facets, and in all cases natural generalisations carry over. We are studying stronger versions of the basic context lemma which hold for all facets. Hitherto we have shown that in the MAY case (which normally has the simplest characterisation, cf. [10]) we need only consider contexts without occurrences of the or combinator.

The basic facet For basic actions, things are very simple and, in fact, only a small finite number of contexts are needed to distinguish the equivalence classes of the MAY and MUST relations. No more than 2 contexts are necessary to distinguish the MAY equivalence classes:

$$[\,]\ \mathsf{trap\ fail}\ , \quad [\,]\ \mathsf{and\ then\ fail}\ .$$

Only 3 contexts suffice for MUST:

$$([\,]\ \mathsf{or\ complete})\ \mathsf{trap\ fail}\ , \quad [\,]\ , \quad ([\,]\ \mathsf{or\ escape})\ \mathsf{and\ then\ fail}\ .$$

This doesn't generalise to other facets.

5 (In)equational action theory

The advantage of having several testing preorders and equivalences, instead of one canonical equivalence, is a greater expressive power. This section demonstrates how the preorders yield interesting inequational characterisations of actions. Furthermore, we discuss how existing equational theory of actions turns into a TEST equational theory.

5.1 Inequational theory

First some properties of the two test preorders which can be shown using the basic context lemma. They complement the existing action laws and further characterise the nondeterministic behaviour of actions. First define $\mathsf{succeed}$, $\mathsf{diverge}$, and the completely nondeterministic action chaos.

$$\mathsf{succeed} = \mathsf{complete\ or\ escape}\ .$$
$$\mathsf{diverge} = \mathsf{unfolding\ unfold}\ .$$
$$\mathsf{chaos} = \left|\begin{array}{l}\mathsf{diverge\ and\ fail\ and\ escape}\\ \end{array}\right.$$
$$\quad\quad \mathsf{trap\ succeed}\ .$$

Below are a sample of inequational laws that hold for actions.

(1) $A_i \ \underset{\sim}{\sqsubset}_{\text{MAY}}\ (A_1 \ \mathsf{or}\ A_2)$.

(2) $(A_1 \ \mathsf{and\ then}\ A_2) \ \underset{\sim}{\sqsubset}_{\text{MAY}}\ (A_1 \ \mathsf{and}\ A_2)$.

(3) $\bot_{\text{MAY}} \ \cong_{\text{MAY}}\ \mathsf{diverge} \ \cong_{\text{MAY}}\ \mathsf{fail}$.

(4) $\top_{\text{MAY}} \ \cong_{\text{MAY}}\ \mathsf{chaos} \ \cong_{\text{MAY}}\ \mathsf{succeed}$.

(5) $(A_1 \ \mathsf{and}\ A_2) \ \underset{\sim}{\sqsubset}_{\text{MUST}}\ (A_1 \ \mathsf{and\ then}\ A_2)$.

(6) $\bot_{\text{MUST}} \ \cong_{\text{MUST}}\ \mathsf{diverge} \ \cong_{\text{MUST}}\ \mathsf{chaos}$.

As a rough intuition of the $\underset{\sim}{\sqsubset}_{\text{MAY}}$ and $\underset{\sim}{\sqsubset}_{\text{MUST}}$ preorders, they both mean 'more divergent', and both $\underset{\sim}{\sqsupset}_{\text{MAY}}$ and $\underset{\sim}{\sqsubset}_{\text{MUST}}$ mean 'less deterministic'. Thus $\mathsf{diverge}$ is both $\bot_{\text{MAY}}$ and $\bot_{\text{MUST}}$, chaos is both $\top_{\text{MAY}}$ and $\bot_{\text{MUST}}$. In this vein, we may read the following lemma may as 'evaluation reduces nondeterminism':

Lemma 5.1 (evaluation) *If* $A_1 \mapsto A_2$ *, then* $A_1 \underset{\sim}{\sqsupset}_{\text{MAY}} A_2$ *and* $A_1 \underset{\sim}{\sqsubset}_{\text{MUST}} A_2$ *.*

Proof: By **basic context lemma** (4.1) and induction on the $\downarrow_{\text{MAY}}$ and $\downarrow_{\text{MUST}}$ predicates, using lemma **choice of execution point** (3.1). □

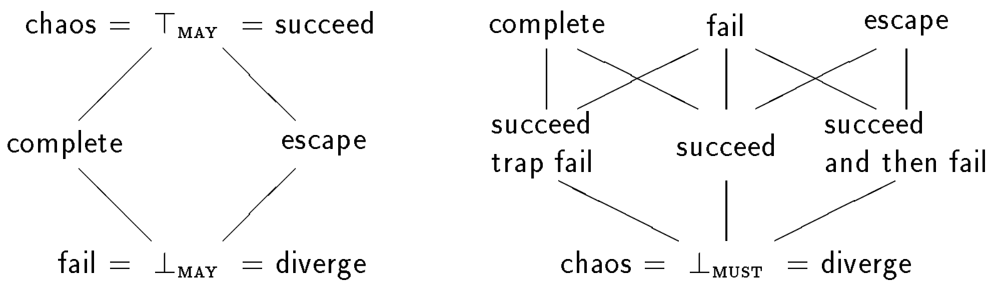
A couple of immediate consequences are:

(7) $\mathsf{complete} \ \underset{\sim}{\sqsubset}_{\text{MAY}}\ (\mathsf{complete\ or}\ A) \ \underset{\sim}{\sqsubset}_{\text{MUST}}\ \mathsf{complete}$.

(8) $\mathsf{fail} \ \underset{\sim}{\sqsubset}_{\text{MAY}}\ (\mathsf{fail\ and}\ A) \ \underset{\sim}{\sqsubset}_{\text{MUST}}\ \mathsf{fail}$.

In deterministic settings, when one term can make a transition into another, then the terms are operationally equivalent [7, 13]. This demonstrates how the presence of nondeterminism weakens action theory.

The basic facet There are, in fact, only 4 MAY and 7 MUST equivalence classes for basic actions. The MAY and MUST orderings can be depicted as follows:



5.2 Equational theory

Under certain conditions we may strengthen the evaluation lemma.

Lemma 5.2 (determined evaluation)
If $A_1 \mapsto A_2$ *is the only transition out of* A_1, *then* $A_1 \cong A_2$.

Proof: As for lemma **evaluation** (5.1). □

Some laws from Section 2.4 are direct consequences of this lemma.

(1) check true $\cong$ complete ; check false $\cong$ fail .
(2) unfolding $A \cong A$ @ unfolding A .
(4a) complete and then $A \cong A$.
(5) escape and then $A \cong$ escape ; fail and then $A \cong$ fail .
(7a) escape trap $A \cong A$.
(8) complete trap $A \cong$ complete ; fail trap $A \cong$ fail .

By the same proof technique as for the proofs of lemma 5.1 and 5.2, we can show all existing basic action laws TEST equivalent.

6 Conclusion

We have defined test preorders for actions and developed a useful operational proof technique for establishing them, based on a tractable alternative characterisation of the test preorders.

The standard range of test preorders and equivalences allowed us to exhibit inequational and equational laws that give rich characterisations of actions. In particular, we have shown that existing action laws hold for TEST equivalence.

Since basic actions are not very expressive, basic action theory is not very interesting in itself. But we believe that all the theory which we have presented for basic actions generalises to full AN. We have developed this for the main parts of the functional, declarative, and imperative facets—we have defined reduction semantics, proved generalisations of lemma 4.1, and proved existing action laws to hold for TEST equivalence. The proof technique used in the basic facet as well as the evaluation lemma 5.1 and 5.2 carry over to these other facets too. All the inequational and equational basic action laws which we have used as illustrations in this paper remain true when other facets are added.

These extensions are all inspired by [11]. We still have not considered the communicative facet but we believe that [1] should be readily applicable.

6.1 Alternative semantics of actions

Above we have reworked the existing action theory in [16] which addresses full AN, based on a nondeterministic structural operational semantics and bisimulation. Alternative semantic foundations for actions exist but none of these appear extensible to a comprehensive theory of full AN. In [5] a categorical semantics is defined for a typed subset of AN and a number of action laws are proven. For a similar subset of AN, [4] defines a versatile natural semantics framework for reasoning about (contextual) semantic equivalences (among many other things). Palsberg [23] and Moura [20] define natural semantics for larger AN subsets. Moura defines a 'functional' action equivalence but not a full action theory.

6.2 Future work

We shall conclude with some of our plans for future work.

Apart from extending our work to full AN, we would like to employ more of Mason and Talcott's framework, viz. develop the equational theory for actions, as in [14], and develop a program logic for actions, as in [11].

We have a number of applications in mind. First of all it would be interesting whether, given an ASD of Mason and Talcott's functional language, we could prove their results by means of our theory.

Bisimulation Bisimulation has been side-stepped in this paper. In many cases, bisimulation offers the simplest proof technique and it definitely deserves study in our setting. Bisimulation is normally the strongest equivalence considered in semantics and is, generally, included in all other equivalences. Standard weak bisimulation doesn't guarantee MUST equivalence, however, because it is insensitive to 'silent' divergence. In [25] this is remedied and, furthermore, weak bisimulation is formulated in terms of preorders. It would be interesting to define such a divergence sensitive weak bisimulation preorder for actions and show it to be included in all testing preorders along the lines in [16, C.4].

In recent years, bisimulation has been subject to much study in functional settings. Cf. e.g. [8] who also summarises research in this area. This line of work may indicate alternative adaptions of (bi)simulation into action theory.

Full abstraction Typically, ASDs are not *fully abstract* [24]. Actions are, as a rule, more expressive than the language being described. Therefore action contexts can distinguish denotations that are indistinguishable by the less expressive language contexts.

This problem is indirectly addressed by our modular approach. In this paper we investigate only basic actions. We have also considered various other subsets of AN: extensions to the functional, declarative, and imperative facets and also restrictions to deterministic subsets of each of these. The ambition is to develop strong theories for subsets of AN in a structured and modular way which can be put together to sound subtheories for arbitrary subsets of AN. If an ASD is not fully abstract, it is, in general, because it doesn't exploit all of AN. If only a subset of AN is used to express the ASD, this subset may have a stronger theory than the weak, universal action theory. This may be a practical way to narrow the full abstraction gap.

Acknowledgements. The presentation of this work has benefited very much from helpful suggestions by Peter Mosses and Peter Ørbæk.

References

[1] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation, Aug. 1994. To appear in *Journal of Functional Programming*.

[2] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34:83–133, 1984.

[3] K.-G. Doh and D. Schmidt. Action semantics-directed prototyping. *Comput. Lang.*, 19(4):213–233, 1993.

[4] K.-G. Doh and D. Schmidt. The facets of action semantics: Some principles and applications. In [17], 1994.

[5] S. Even and D. A. Schmidt. Category sorted algebra-based action semantics. *Theoretical Comput. Sci.*, 77:73–96, 1990.

[6] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*. IFIP, 1987.

[7] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Comput. Sci.*, 103:235–271, 1992.

[8] A. D. Gordon. A tutorial on co-induction and functional programming. In *Glasgow Workshop on Functional Programming*, 1994.

[9] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In [17], 1994.

[10] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[11] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects, 1993. To appear in *Information and Computation*.

[12] S. B. Lassen. Design and Semantics of Action Notation. In [17], 1994.

[13] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):297–327, 1991.

[14] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Comput. Sci.*, 105(2):167–215, 1993.

[15] P. D. Mosses. Abstract semantic algebras! In D. Bjørner, editor, *Formal Description of Programming Concepts II*. IFIP, 1983.

[16] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[17] P. D. Mosses, editor. *Proceedings of the First International Workshop on Action Semantics (Edinburgh, Scotland, April 1994)*, number NS-94-1 in BRICS Notes Series. Dept. of Computer Science, Univ. of Aarhus, 1994.

[18] P. D. Mosses. Unified algebras and abstract syntax. In *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[19] P. D. Mosses and D. A. Watt. Pascal action semantics, version 0.6. Available by FTP from ftp.daimi.aau.dk in pub/action/pascal, Mar. 1993.

[20] H. Moura. *Action Notation Transformations*. Ph.D. thesis, University of Glasgow, 1993.

[21] H. Moura and D. Watt. Action transformations in the ACTRESS compiler generator. In *CC'94*, volume 786 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[22] P. Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In *CC'94*, volume 786 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[23] J. Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Dept. of Computer Science, Univ. of Aarhus, 1992.

[24] G. D. Plotkin. LCF considered as a programming language. *Theoretical Comput. Sci.*, 5:223–255, 1977.

[25] D. J. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.