Math. Struct. in Comp. Science (1997), *vol.* 7, *pp.* 557–589. Printed in the United Kingdom
© 1997 Cambridge University Press

Action semantics reasoning about functional programs

S. B. LASSEN[†]

BRICS[‡], *Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark*

email: thales@brics.dk

Received 1 December 1995; revised September 1996

This paper develops the functional part of a theory of action semantics for reasoning about programs. Action notation, the specification language of action semantics, is given an evaluation semantics, and operational techniques from process theory and functional programming are applied in the development of a versatile action theory. The power of the theory is demonstrated by means of action semantic proofs of functional program equivalences.

1. Introduction

Action semantics is a semantic description framework with very good pragmatic properties (Mosses 1992; Mosses 1996). It has been used to formalise a wide range of realistic programming languages, as demonstrated by complete descriptions of Pascal (Mosses and Watt 1993), ANDF (Hansen and Toft 1994), Standard ML (Watt 1996) (in progress), most of Occam2 (Buhl 1994), as well as numerous other procedural, object-oriented, and parallel languages. Moreover, a number of compiler generators have been based on action semantics, *e.g.* (Moura and Watt 1994; Ørbæk 1994).

A useful theory for reasoning about programs in action semantics has a wide practical scope. A strong action theory would offer techniques for reasoning about programs in any programming language that can be described in action semantics.

The good modularity and extensibility of action semantic descriptions derive from the unique design of the specification language, called action notation. It is a comparatively large semantic metalanguage with notation for all the fundamental computational concepts found in programming languages. In Lassen (1995) we developed an action theory for a small control fragment (the basic facet) of action notation. Here we extend this work to a substantial part of action notation (the functional and declarative facets) sufficient for describing functional programming languages. The computational features we deal

[†] Supported by the Danish National Science Research Council.
[‡] Centre for Basic Research in Computer Science, Danish National Research Foundation.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

with include unbounded nondeterminism, higher-order functions, bindings with dynamic scope, and data processing on abstract data types.

The results we present contribute both to the theory of action semantics and to ongoing research on operational reasoning about higher-order programming languages.

Based on an evaluation semantics for action notation, we define a collection of operational semantic preorders and equivalences for action notation. We develop a range of powerful techniques for reasoning about these preorders and equivalences and apply this theory to prove functional program equivalences based on an action semantic description of a functional language.

Our reasoning techniques are based on operational theory for functional programming languages and for process calculi. We apply the applicative bisimulation approach (Abramsky 1990) to the combinator-based action notation. We combine (unbounded) nondeterminism and abstract data types with applicative bisimulation, and present a co-inductive simulation proof method for contextual testing preorders in a nondeterministic setting. We also sketch how the CIU approach of Mason and Talcott (1991) applies to actions.

1.1. *Outline.*

In Section 2 we introduce action semantics by giving an action semantic description of a small functional language, which we use to illustrate our reasoning techniques in Section 7. Section 3 presents a functional/declarative subset of action notation and in Section 4 an operational semantics for actions is defined. Operational semantic preorders and equivalences are defined in Section 5 and their properties are investigated. In Section 6 we develop co-inductive simulation techniques for reasoning about the semantic preorders, and a number of (in)equational laws and inference rules for action notation are established. These are applied in Section 7 to show various program equivalences for functional programs. Finally, Section 8 discusses related and future work. Appendix A summarises the unified algebra framework.

2. Example functional language

As illustration of action semantics, let us look at an action semantic description of a simple functional language – a kind of 'untyped PCF over Booleans', *i.e.* an untyped, call-by-name λ-calculus extended with explicit recursion, Boolean constants and **if-then-else**. For example, ($\lambda x.$**if** x **then false else true**) is a Boolean negation function and $\Omega \stackrel{\text{def}}{=}$ **rec** $x.x$ is a divergent program.

The action semantic description defines the abstract syntax of the language and specifies semantic functions that maps abstract syntax into meaning. The grammar and semantic functions are specified in the unified algebra formalism, summarised in Appendix A.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

2.1. *Abstract syntax*

The algebraic definition of abstract syntax trees given below can, more or less, be read as a BNF grammar. Emphatic brackets, ⟦…⟧, indicate nodes in an abstract syntax tree.

grammar:

- Expression = Identifier | "**true**" | "**false**" |
  ⟦ "λ" Identifier "." Expression ⟧ |
  ⟦ Expression Expression ⟧ |
  ⟦ "**rec**" Identifier "." Expression ⟧ |
  ⟦ "**if**" Expression "**then**" Expression "**else**" Expression ⟧ .
- Identifier = ⟦ letter^{+} ⟧ .

2.2. *Semantic functions*

Action semantic descriptions are syntax-directed in the denotational style: compositional semantic functions map abstract syntax into meaning and are defined inductively by semantic equations. There is one universal semantic domain, namely action, the sort of actions. Actions are expressed in a notation that looks a little like informal English prose but is, in fact, a completely formal combinator-based notation. The verbose notation should be suggestive of the meaning of the action denotations. Our discussion of the semantic equations given below provides some hints. After the formalisation of actions in Section 4 the reader may want to return and study the semantic equations in more detail.

- evaluate _ :: Expression → action .

(1) evaluate I:Identifier = enact the data bound to I .

In this language, variable identifiers are bound to suspended evaluations, represented by abstractions, that are enacted when variables are looked up.

(2) evaluate "**true**" = give true .

(3) evaluate "**false**" = give false .

The "give" primitive puts data on the transient data flow.

(4) evaluate ⟦ "λ" I:Identifier "." E:Expression ⟧ =
give closure abstraction of (furthermore bind I to given data
hence evaluate E) .

Functions are represented by abstractions. The "closure" operation closes the abstraction up with the current variable bindings and thus provides static scope. Upon invocation, the bound variable is bound to the input data; "furthermore" extends the variable bindings of the closure with this extra binding; and the compound bindings are passed into the function body by "hence".

(5) evaluate ⟦ E_1:Expression E_2:Expression ⟧ =
evaluate E_1 then enact (application of given data
to closure abstraction of evaluate E_2) .

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

In function application, the operator is first evaluated to a function; it is then applied to the operand whose evaluation is suspended in an abstraction. This models a call-by-name parameter mechanism.

(6) evaluate ⟦ "**rec**" *I*:Identifier "." *E*:Expression ⟧ =
    unfolding (furthermore bind *I* to closure abstraction of unfold
        hence evaluate *E*) .

The body of the **rec** expression is evaluated in variable bindings recursively extended with (the suspended evaluation of) the **rec** expression. "unfolding" is a recursion combinator and "unfold" marks a recursive occurrence.†

(7) evaluate ⟦ "**if**" E_0:Expression "**then**" E_1:Expression "**else**" E_2:Expression ⟧ =
    evaluate E_0 then
    ((check given data and then evaluate E_1)
      or
      (check not given data and then evaluate E_2)) .

The combinator "then" passes the result of the evaluation of the condition into a choice construct. Each branch checks that the result is true (respectively, false), and then evaluates the corresponding expression; otherwise the branch fails and the "or" combinator chooses the alternative branch.

(See Doh (1993) for a more complete introduction to action semantics of functional languages.)

3. Action notation

This section describes the functional/declarative subset of action notation and the underlying data types used in the action semantic description above.

Action notation has a two-sorted syntax: actions and yielders. Actions describe control and information flow. Yielders are unevaluated data that may depend on current information.

3.1. *Actions*

Actions either complete, fail or diverge. Completing corresponds to normal termination and returns resulting information. Failing is abnormal termination. The functional/declarative subset of action notation processes two kinds of information, namely transient data and bindings.

Actions are either primitives or compounds built using combinators. Some primitives are parameterised by yielders. The following algebraic specification of sort action may be read as a grammar with the proper sorts action, yielder and data as nonterminals (see Appendix A).

† Those familiar with action semantics will notice that our description of recursion deviates from the standard approach of action semantics. We do this in order to deal with recursive bindings in purely declarative terms.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

- action = unfold | unfolding action | check yielder | enact yielder | action or action | action and action | action and then action | give yielder | choose data | action then action | produce yielder | action hence action | furthermore action .

We let metavariable PRIM$_0$ range over the 0-ary primitives "unfold" and "choose D", for every $D \leqslant$ data (*i.e.*, D is a subsort of data, defined below, so "choose D" is a sort indexed family of primitives).

- PRIM$_0$: action .

PRIM$_1$ ranges over the unary primitives "check", "enact", "give" and "produce" that take a yielder argument.

- PRIM$_1$ _ :: yielder → action (*total*, *injective*) .

Finally, UNARY and BINARY range over action combinators. "unfolding" and "furthermore" are unary combinators. The binary infix combinators are "or", "and", "and then", "then", and "hence".

- UNARY _ :: action → action (*total*, *injective*) .
- _ BINARY _ :: action, action → action (*total*, *injective*) .

We write sort action more succinctly as

- action = PRIM$_0$ | PRIM$_1$ yielder | UNARY action | action BINARY action .

3.2. *Yielders*

Yielders are unevaluated data that may depend on current information via additional constructs for accessing incoming transient data and bindings:

- given data, current bindings : yielder .

Let DATA-OP$_n$ range over n-ary data operations, for $n \geqslant 0$.

- DATA-OP$_n$(_ ... _) :: data,..., data → data (*partial*) .

They are all overloaded to act as constructors for compound yielders.

- DATA-OP$_n$(_ ... _) :: yielder,..., yielder → yielder (*partial*) .

(And the extension is total and injective for proper yielder arguments, *i.e.* yielders that are not data.)

We can specify sort yielder as

- yielder = data | given data | current bindings | DATA-OP$_n$(yielder ... yielder) .

3.3. *Information flow*

Transient data and bindings form two orthogonal information flows. The corresponding data types, data and bindings, are defined in Section 3.6.

Data includes all processable data types: truth values, abstractions, bindings, *etc.* Transient data is constructed using the "give" and "choose" action primitives, is passed on by the "then" combinator, and is accessed by the yielder "given data". The action

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

"regive" that passes through incoming transient data can be expressed as

regive $\stackrel{\text{def}}{=}$ give given data .

Higher-order computation is expressed in terms of abstraction and enaction: actions may be incorporated in data ('reified') and processed as data via the data type abstraction; and actions in abstractions are enacted ('reflected') by the "enact" primitive.

Bindings are installed by the "produce" primitive, are passed on by the "hence" combinator, and a copy is retrieved by "current bindings". The action "rebind" passes through incoming bindings:

rebind $\stackrel{\text{def}}{=}$ produce current bindings .

"furthermore A" extends the incoming bindings with those produced by action A. The standard abbreviations "bind Y to Y'" and "the data bound to Y" for pointwise operations on bindings are defined in Section 3.6.2 below. Copying and installation of bindings is a flexible mechanism that can be used to provide both static and dynamic scope. "closure A", defined in Section 3.6.3, builds a closure of an abstraction by wrapping it up with a copy of current bindings.

3.4. *Recursion*

"unfolding" is the action recursion combinator and the place-holder "unfold" marks recursive unfoldings in its textual scope. For instance, a divergent action is most shortly written as:

diverge $\stackrel{\text{def}}{=}$ unfolding unfold .

An action A is *unfold-closed* if it has no free occurrences of the primitive "unfold", *i.e.* occurrences not enclosed by the unary combinator "unfolding". "unfold" can occur free in abstractions, therefore abstractions, data and yielders can also have free occurrences of "unfold" or be unfold-closed.†

"unfold" is only computationally meaningful when it is enclosed by "unfolding". Only unfold-closed actions can be performed.

The auxiliary substitution operator _@_ (not itself part of action notation) substitutes its second argument for all free occurrences of "unfold" in the first argument, (A_1@A_2 ="$A_1[A_2/\text{unfold}]$").

- _ @ _ :: action, action → action ,
  yielder, action → yielder (*associative*, *unit is* unfold) .

"unfolding A" may be thought of as the infinite action term "A@A@A@…".

3.5. *Nondeterminism*

There are two sources of nondeterminism in the language of actions: bounded nondeterminism due to the binary choice combinator "or"; and unbounded nondeterminism due

† Normally (Mosses 1992), the scope of "unfolding" does not extend into the bodies of abstractions. Our definition can be made to coincide with the usual definition by restricting abstraction to unfold-closed actions. However, our treatment enhances the semantics of "unfolding" in a useful way. In particular, it makes our description of recursion in Section 2.2(6) possible.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

to the infinitely branching primitive "choose", which selects an arbitrary individual from a possibly infinite sort.

The "or" combinator has an 'angelic' flavour: it chooses a non-failing argument, if any, and can model guarded choice, as exploited in the (deterministic) description of **if-then-else** in Section 2. We restrict failure to one source, namely "check Y" when Y evaluates to false; if Y evaluates to true, "check Y" completes with empty information.

In Section 7.2 we need an 'erratic' binary choice action combinator, "erratic or", to model a nondeterministic extension of our functional language. It is possible to derive this combinator from "or" and "choose".

A_1 erratic or A_2 $\stackrel{\text{def}}{=}$
(choose truth-value and regive) then
((check first of given data and then (give rest of given data then A_1))
or
(check not first of given data and then (give rest of given data then A_2))).

A truth value is chosen and "or" dispatches control accordingly to action A_1 or A_2 (the unchosen branch fails). Some bookkeeping is performed to pass the initial transient data to A_1 and A_2. The "and" combinator tuples the data from its two subactions – here a truth value and the incoming transient data – and "first of given data" and "rest of given data" splits the data tuple at the first item. The "and" combinator evaluates its subactions in any order, whereas "and then" sequences evaluates left-to-right.

As another illustration of action constructs, we consider how countable nondeterminism can be approximated by means of binary choice and recursion. For example, let

generate $\stackrel{\text{def}}{=}$ unfolding (give 0 or (unfold then give successor of given data)) .

"generate" gives an arbitrary natural number or diverges, and is thus equivalent to "choose natural", except that the latter always terminates. (natural is the sort of natural numbers.)

3.6. *Data*

The syntax and semantics of actions and yielders depend on the underlying abstract data types data, bindings and abstraction.

3.6.1. *Data.* Data are associative tuples of elements of sort datum.

- data = () | (datum, data) .
- _ , _ :: data, data → data (*total*, *associative*, *unit is* ()) .
- first of _ :: data → datum (*partial*) .
- rest of _ :: data → data (*partial*) .

(1) $d = (d_1$:datum, d_2:data) ⇒ first of $d = d_1$; rest of $d = d_2$.

Sort datum consists of truth values, tokens, bindings and abstractions, plus application-specific data types (numbers, text strings, records, *etc.*).

- datum ⩾ truth-value | token | bindings | abstraction .

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

The inequation means that datum may include other data types than these standard ones.

- truth-value = true | false (*individual*) .
- not _ :: truth-value → truth-value (*total*, *injective*) .
- when _ then _ :: true, data → data (*total*, *injective*) .

(2) not true = false ; not false = true .

(3) d : data ⇒ when true then $d = d$.

Sort token is application-specific but is required to have a truth-valued equality operation "is".

- _ is _ :: token, token → truth-value (*total*) .

In the example action semantic description in Section 2, tokens were assumed to be text strings; no other application-specific data types were introduced.

As a simple example of an application-specific data type, suppose natural numbers are added to our functional language. Then we introduce the corresponding data type natural (specified in Appendix A).

- natural ⩽ data .

"0" and "successor of" become 0-ary and unary partial data operations, respectively.

3.6.2. *Bindings.* Bindings are essentially finite maps from tokens to data.

- {} : bindings .
- {_ ↦ _} :: token, data → bindings (*total*) .
- merge(_,_) :: bindings, bindings → bindings (*total*, *associative*, *commutative*, *unit is* {}) .
- overlay(_,_) :: bindings, bindings → bindings (*total*, *associative*, *idempotent*, *unit is* {}) .
- _ at _ :: bindings, token → data (*partial*) .

$\{x \mapsto d\}$ binds x to d, merge(b_1, b_2) is the union of bindings b_1, b_2 (clashing tokens become inaccessible), overlay(b_1, b_2) overlays b_1 over b_2, and b at x looks up x in b.

The algebraic properties of merge and overlay, given above, together with the following equations (1)–(2) rewrite bindings b to the form b=overlay($\{x \mapsto d\}$, b') whenever b contains a binding of x to d and this binding is not merged with any other binding of x. (The proof of this is not trivial.) Then tokens are looked up in bindings of that form, (3).

x, x' : token ; d, d' : data ; b, b' : bindings ⇒

(1) x is x' = false ⇒ overlay($\{x \mapsto d\}$, $\{x' \mapsto d'\}$) = merge($\{x \mapsto d\}$, $\{x' \mapsto d'\}$) ;

(2) overlay($\{x \mapsto d\}$, b') = merge($\{x \mapsto d\}$, b') ⇒
overlay(b', $\{x \mapsto d\}$) = merge(b', $\{x \mapsto d\}$) ;
merge(overlay($\{x \mapsto d\}$, b), b') = overlay($\{x \mapsto d\}$, merge(b, b')) ;
overlay(merge($\{x \mapsto d\}$, b), b') = merge($\{x \mapsto d\}$, overlay(b, b')) ;

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

(3) x is $x' =$ true $\Rightarrow$ overlay($\{x \mapsto d\}$, b) at $x' = d$.

The actions and yielders for manipulating individual tokens are expressed in terms of the corresponding notation for processing entire bindings:

bind Y to Y' $\stackrel{\text{def}}{=}$ produce $\{Y \mapsto Y'\}$.

the data bound to Y $\stackrel{\text{def}}{=}$ current bindings at Y .

3.6.3. *Abstractions.* Abstractions are data that incorporate actions.

- abstraction of _ :: action → abstraction (*total*, *injective*) .

"abstraction of" is not a data operation but the abstraction building operations below are. All action combinators other than "unfolding" are overloaded to combine abstractions. ("unfolding" is excluded, since otherwise clause (1) would conflict with textual scope for "unfold".)

- UNARY _ :: abstraction → abstraction (*total*, *injective*) .
- _ BINARY _ :: abstraction, abstraction → abstraction (*total*, *injective*) .

A, A' : action $\Rightarrow$

(1) UNARY abstraction of A = abstraction of UNARY A ;

(2) abstraction of A BINARY abstraction of A' = abstraction of (A BINARY A') .

Similarly, there are abstraction building operations corresponding to the "give" and "produce" action primitives.

- provision of _ :: data → abstraction (*total*, *injective*) .
- production of _ :: bindings → abstraction (*total*, *injective*) .

(3) d : data $\Rightarrow$ provision of d = abstraction of give d .

(4) b : bindings $\Rightarrow$ production of b = abstraction of produce b .

The abstraction building operations can be used to bundle abstractions with transient data and bindings to be supplied upon enaction. The following standard yielders (used in the action semantic description in Section 2) dynamically supply data and bindings to abstractions.

application of Y to Y' $\stackrel{\text{def}}{=}$ (provision of Y') then Y .

closure Y $\stackrel{\text{def}}{=}$ (production of current bindings) hence Y .

3.6.4. *Assumptions.* The generality of unified algebras makes it possible to specify subtle relationships between data individuals, sorts and data operations. In order to obtain a useful theory about actions, we make the following assumptions about data and data operations:

1 Every data operation DATA-OP$_n$ is *partial* and *discrete*. Here partial means that individuals are mapped to individuals or to vacuous sorts (representing undefined outcome).

$$d_1, \ldots, d_n : \text{data} \; ; \; d : \text{DATA-OP}_n(d_1 \ldots d_n) \Rightarrow \text{DATA-OP}_n(d_1 \ldots d_n) : \text{data} .$$

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

And discrete means that only individuals are mapped to individuals.

$$\text{DATA-OP}_n(d_1 \ldots d_n) : \mathsf{data} \;\Rightarrow\; d_1, \ldots, d_n : \mathsf{data} \; .$$

Therefore we do not classify sort union, " | ", as a data operation.

2 Every data individual d can be written as a term generated by the grammar:

$$d \quad ::= \quad \mathsf{abstraction\ of}\ A \quad | \quad \text{DATA-OP}_n(d_1 \ldots d_n) \; , \tag{1}$$

where A ranges over action terms, A:action, and $n \geqslant 0$.

3 We exclude data operations that operate on the intension (inner structure) of abstractions. Only upon enaction are abstractions opened up. Data operations must only manipulate abstractions in a *uniform* way as 'black boxes'.

Assumptions 1 and 2 rule out the use of proper sorts as values in the specification of data. To illustrate Assumption 3, consider some examples of data operations that violate the uniformity requirement:

- is-and _ :: abstraction → truth-value (*total*) .
- and-left _ :: abstraction → abstraction (*partial*) .

a_1, a_2 : abstraction ⇒

(1) is-and (a_1 or a_2) = false ; is-and (a_1 and a_2) = true ; ...

(2) and-left (a_1 and a_2) = a_1 .

"is-and" and "and-left" are not uniform in abstractions as they expose the syntax of abstracted actions. On the other hand, the following test for whether a datum is an abstraction is uniform because it does not inspect the contents of abstractions:

- is-abstraction _ :: datum → truth-value (*total*) .

(1) a : abstraction ⇒ is-abstraction a = true .

(2) d : truth-value | token | bindings ⇒ is-abstraction d = false .

The formalisation of uniformity is quite delicate and we postpone it to Section 5.2.

4. Evaluation semantics

We formalise the meaning of actions by means of an evaluation semantics. It is a big-step, structural operational semantics. The exact formalisation of the semantics of actions can be done in a number of ways. Compared to Mosses' original small-step structural operational semantics for action notation in Mosses (1992, Appendix C), the present evaluation semantics offers some technical advantages for the development of the theory in Section 6. The two operational semantics are equivalent, except for the minor deviations discussed in the presentation of action notation above.

4.1. *Yielder evaluation*

First we define evaluation of yielders to data as an environment indexed relation between yielders and data, given by judgements of the form, $i \vdash Y \Downarrow d$, where i : info; Y : yielder; d : data. The environment i is the incoming transient data and bindings.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

- info = (data, bindings) .

The evaluation substitutes these for all occurrences of "given data" and "current bindings".

d : data ; b : bindings $\Rightarrow$

(1) $(d, b) \vdash$ given data $\Downarrow d$; $(d, b) \vdash$ current bindings $\Downarrow b$.

i : info ; $d_1, \ldots, d_n$: data ; $Y_1, \ldots, Y_n$: yielder ; $i \vdash Y_1 \Downarrow d_1$; ...; $i \vdash Y_n \Downarrow d_n$ $\Rightarrow$

(2) DATA-OP$_n(d_1 \ldots d_n)$: data $\Rightarrow$ $i \vdash$ DATA-OP$_n(Y_1 \ldots Y_n) \Downarrow$ DATA-OP$_n(d_1 \ldots d_n)$.

Yielder evaluation is deterministic:

$i \vdash Y \Downarrow d$; $i \vdash Y \Downarrow d'$ $\Rightarrow$ $d = d'$.

4.2. *Action evaluation*

The evaluation semantics is an environment indexed relation between unfold-closed actions A and outcomes t, written $i \vdash A \Downarrow^{\text{MAY}} t$, with i : info; A : action; t : terminated. Outcomes are either completed with resulting data and bindings, or failed.

- terminated = completed | failed .
- _×_ :: data, bindings → completed (*total*, *injective*) .
- failed : terminated .

Evaluation is nondeterministic. We say an action *may terminate* if it evaluates to some outcome.

i : info ; Y : yielder ; $i \vdash Y \Downarrow d$ $\Rightarrow$

(1) $d =$ true $\Rightarrow$ $i \vdash$ check $Y \Downarrow^{\text{MAY}} ()\times\{\}$; $d =$ false $\Rightarrow$ $i \vdash$ check $Y \Downarrow^{\text{MAY}}$ failed ;

(2) $d =$ abstraction of A ; $((),\{\}) \vdash A \Downarrow^{\text{MAY}} t$ $\Rightarrow$ $i \vdash$ enact $Y \Downarrow^{\text{MAY}} t$;

(3) d : data $\Rightarrow$ $i \vdash$ give $Y \Downarrow^{\text{MAY}} d\times\{\}$;

(4) d : bindings $\Rightarrow$ $i \vdash$ produce $Y \Downarrow^{\text{MAY}} ()\times d$;

(5) $D \leqslant$ data ; $d : D$ $\Rightarrow$ $i \vdash$ choose $D \Downarrow^{\text{MAY}} d\times\{\}$.

d, d_1, d_2 : data ; b, b_1, b_2 : bindings ; $A, A_1, A_2, A_2', A_2'', B_1, B_2, B_2', B_2''$: action ;
$i = (d, b)$; $i \vdash A_1 \Downarrow^{\text{MAY}} d_1\times b_1$; $i \vdash B_1 \Downarrow^{\text{MAY}}$ failed ; $i \vdash A_2 \Downarrow^{\text{MAY}} d_2\times b_2$;
$i \vdash B_2 \Downarrow^{\text{MAY}}$ failed ;
$i' = (d_1, b)$; $i' \vdash A_2' \Downarrow^{\text{MAY}} d_2\times b_2$; $i' \vdash B_2' \Downarrow^{\text{MAY}}$ failed ;
$i'' = (d, b_1)$; $i'' \vdash A_2'' \Downarrow^{\text{MAY}} d_2\times b_2$; $i'' \vdash B_2'' \Downarrow^{\text{MAY}}$ failed $\Rightarrow$

(6) $i \vdash A_1$ or $A \Downarrow^{\text{MAY}} d_1\times b_1$; $i \vdash A$ or $A_2 \Downarrow^{\text{MAY}} d_2\times b_2$; $i \vdash B_1$ or $B_2 \Downarrow^{\text{MAY}}$ failed ;

(7) $i \vdash A_1$ and $A_2 \Downarrow^{\text{MAY}} (d_1, d_2)\times$merge$(b_1, b_2)$; $i \vdash B_1$ and $A \Downarrow^{\text{MAY}}$ failed ;
$i \vdash A$ and $B_2 \Downarrow^{\text{MAY}}$ failed ;

(8) $i \vdash A_1$ and then $A_2 \Downarrow^{\text{MAY}} (d_1, d_2)\times$merge$(b_1, b_2)$; $i \vdash B_1$ and then $A \Downarrow^{\text{MAY}}$ failed ;
$i \vdash A_1$ and then $B_2 \Downarrow^{\text{MAY}}$ failed ;

(9) $i \vdash A_1$ then $A_2' \Downarrow^{\text{MAY}} d_2\times$merge$(b_1, b_2)$; $i \vdash B_1$ then $A \Downarrow^{\text{MAY}}$ failed ;
$i \vdash A_1$ then $B_2' \Downarrow^{\text{MAY}}$ failed ;

(10) $i \vdash A_1$ hence $A_2'' \Downarrow^{\text{MAY}} (d_1, d_2)\times b_2$; $i \vdash B_1$ hence $A \Downarrow^{\text{MAY}}$ failed ;
$i \vdash A_1$ hence $B_2'' \Downarrow^{\text{MAY}}$ failed ;

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

(11) $i \vdash \mathsf{furthermore}\ A_1 \Downarrow^{\mathrm{MAY}} d_1 \times \mathsf{overlay}(b_1, b)\ ;\ i \vdash \mathsf{furthermore}\ B_1 \Downarrow^{\mathrm{MAY}} \mathsf{failed}\ .$

(12) $i : \mathsf{info}\ ;\ A : \mathsf{action}\ \Rightarrow\ i \vdash A@\mathsf{unfolding}\ A \Downarrow^{\mathrm{MAY}} t\ \Rightarrow\ i \vdash A \Downarrow^{\mathrm{MAY}} t\ .$

For example, for all *i*:info; *n*:natural, we have $i \vdash \mathsf{choose\ natural} \Downarrow^{\mathrm{MAY}} n \times \{\}$, by evaluation rule (5), and likewise for "generate" (Section 3.5), $i \vdash \mathsf{generate} \Downarrow^{\mathrm{MAY}} n \times \{\}$, which follows by induction on *n* and involves evaluation rules (12), (6), (3), and (9).

By inspection of evaluation rules (3), (9), (4), and (10), we see that the information in the environment can be represented syntactically as follows:

$$\begin{array}{rcl}(d, b) \vdash A \Downarrow^{\mathrm{MAY}} t & \Leftrightarrow & (d', b) \vdash \mathsf{give}\ d\ \mathsf{then}\ A \Downarrow^{\mathrm{MAY}} t \\ & \Leftrightarrow & (d, b') \vdash \mathsf{produce}\ b\ \mathsf{hence}\ A \Downarrow^{\mathrm{MAY}} t\ ,\end{array} \tag{2}$$

for all d, d' : data; b, b' : bindings; A : action; t : terminated.

This big-step evaluation semantics is more abstract than a small-step operational semantics. It only describes terminating computations and abstracts from divergence. Small steps give more intensional information, *viz.* the intermediate configurations of a computation. They describe, for instance, how the 'symmetric' combinators "or" and "and" may interleave their arguments.

4.3. *Termination*

Apart from what an action *may* evaluate to, we are going to need complementary information about termination, namely whether an action *must terminate*. This can be expressed very simply in terms of the possible reduction sequences of a small-step operational semantics. A big-step definition becomes more elaborate (and more informative) as it provides information that the associated evaluation semantics abstracts from.

We write $i \vdash A \Downarrow^{\mathrm{MUST}}$ to mean that unfold-closed action A *must terminate* in environment i. For the deterministic constructs, the definition is close to that of evaluation above. (The following definition of must termination involves quantifications going beyond the Horn clause logic of unified algebras. However, it is a valid inductive definition, because all axioms are monotone in the must terminate predicate.)

$i : \mathsf{info}\ ;\ Y : \mathsf{yielder}\ ;\ i \vdash Y \Downarrow d\ \Rightarrow$

(1) $\mathsf{d} : \mathsf{truth\text{-}value}\ \Rightarrow\ i \vdash \mathsf{check}\ Y \Downarrow^{\mathrm{MUST}}\ ;$

(2) $d = \mathsf{abstraction\ of}\ A\ ;\ A : \mathsf{action}\ ;\ ((),\{\}) \vdash A \Downarrow^{\mathrm{MUST}}\ \Rightarrow\ i \vdash \mathsf{enact}\ Y \Downarrow^{\mathrm{MUST}}\ ;$

(3) $d : \mathsf{data}\ \Rightarrow\ i \vdash \mathsf{give}\ Y \Downarrow^{\mathrm{MUST}}\ ;$

(4) $d : \mathsf{bindings}\ \Rightarrow\ i \vdash \mathsf{produce}\ Y \Downarrow^{\mathrm{MUST}}\ ;$

(5) $D \leqslant \mathsf{data}\ ;\ d : D\ \Rightarrow\ i \vdash \mathsf{choose}\ D \Downarrow^{\mathrm{MUST}}\ .$

d, d_1 : data ; b, b_1 : bindings ; $i = (d, b)$; A_1, A_2 : action $\Rightarrow$

(6) $i \vdash A_1 \Downarrow^{\mathrm{MUST}}\ ;\ i \vdash A_2 \Downarrow^{\mathrm{MUST}}\ \Rightarrow\ i \vdash A_1\ \mathsf{or}\ A_2 \Downarrow^{\mathrm{MUST}}\ ;$

(7) $i \vdash A_1 \Downarrow^{\mathrm{MUST}}\ ;\ i \vdash A_2 \Downarrow^{\mathrm{MUST}}\ \Rightarrow\ i \vdash A_1\ \mathsf{and}\ A_2 \Downarrow^{\mathrm{MUST}}\ ;$

(8) $i \vdash A_1 \Downarrow^{\mathrm{MUST}}\ ;\ (\forall d_1, b_1 \mid i \vdash A_1 \Downarrow^{\mathrm{MAY}} d_1 \times b_1)\ i \vdash A_2 \Downarrow^{\mathrm{MUST}}\ \Rightarrow$
$i \vdash A_1\ \mathsf{and\ then}\ A_2 \Downarrow^{\mathrm{MUST}}\ ;$

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

(9) $i \vdash A_1 \Downarrow^{\text{MUST}}$; $(\forall d_1, b_1 \mid i \vdash A_1 \Downarrow^{\text{MAY}} d_1 \times b_1)\ (d_1, b) \vdash A_2 \Downarrow^{\text{MUST}} \Rightarrow$
$i \vdash A_1$ then $A_2 \Downarrow^{\text{MUST}}$;

(10) $i \vdash A_1 \Downarrow^{\text{MUST}}$; $(\forall d_1, b_1 \mid i \vdash A_1 \Downarrow^{\text{MAY}} d_1 \times b_1)\ (d, b_1) \vdash A_2 \Downarrow^{\text{MUST}} \Rightarrow$
$i \vdash A_1$ hence $A_2 \Downarrow^{\text{MUST}}$;

(11) $i \vdash A_1 \Downarrow^{\text{MUST}} \Rightarrow i \vdash$ furthermore $A_1 \Downarrow^{\text{MUST}}$.

(12) i : info ; A : action $\Rightarrow i \vdash A$@unfolding $A \Downarrow^{\text{MUST}} \Rightarrow i \vdash A \Downarrow^{\text{MUST}}$.

Termination rules (8)–(10) for the sequential combinators require both that the first subaction must terminate and, if it completes, that the second subaction must terminate. For example, $i \vdash$ check false and then diverge $\Downarrow^{\text{MUST}}$, because this action will always fail. But not $i \vdash$ check false and diverge $\Downarrow^{\text{MUST}}$ since the non-sequential "and" combinator may attempt to evaluate its second subaction, which diverges.

Must termination distinguishes "choose natural" and "generate". Termination rule (5) gives $i \vdash$ choose natural $\Downarrow^{\text{MUST}}$. In contrast, $i \vdash$ generate $\Downarrow^{\text{MUST}}$ is impossible because any derivation of this requires itself as a premise. This is in correspondence with our operational understanding that "generate" can diverge.

The expected relationship between must termination and evaluation

$$i \vdash A \Downarrow^{\text{MUST}} \quad \text{implies} \quad (\exists t)\ i \vdash A \Downarrow^{\text{MAY}} t$$

follows by induction on the derivation of $i \vdash A \Downarrow^{\text{MUST}}$.

The evaluation relation and must termination predicate specify when actions may or must terminate. We can derive *may complete* and *must complete* predicates, $i \vdash A \downarrow_{\text{MAY}}$ and $i \vdash A \downarrow_{\text{MUST}}$, which tell whether actions may or must complete.

$$i \vdash A \downarrow_{\text{MAY}} \stackrel{\text{def}}{\Leftrightarrow} (\exists t \mid i \vdash A \Downarrow^{\text{MAY}} t)\ \ t : \text{completed} .$$
$$i \vdash A \downarrow_{\text{MUST}} \stackrel{\text{def}}{\Leftrightarrow} i \vdash A \Downarrow^{\text{MUST}} \wedge (\forall t \mid i \vdash A \Downarrow^{\text{MAY}} t)\ \ t : \text{completed} .$$

Syntactic representation of the environment, (2), also holds for the termination and completion predicates.

5. Action equivalence

Based on the evaluation semantics for actions, we shall now study semantic equivalence on actions. First we fix some terminology about relations on actions and formalise the uniformity requirement for data operations from Section 3.6.4. Then we define semantic preorders and equivalences on actions, and investigate their properties.

5.1. *Compatible relations*

For every binary relation R on actions, its *compatible refinement* (Gordon 1995b), $\widehat{R}$, relates action terms with identical outermost syntactic constructor and immediate subterms

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

pairwise related by R, as given by the rules:

$$\frac{}{\textsc{prim}_0 \;\widehat{R}\; \textsc{prim}_0} \qquad \frac{Y \;\widehat{R}\; Y'}{\textsc{prim}_1\ Y \;\widehat{R}\; \textsc{prim}_1\ Y'}$$

$$\frac{A \;R\; A'}{\textsc{unary}\ A \;\widehat{R}\; \textsc{unary}\ A'} \qquad \frac{A_1 \;R\; A_1' \,,\quad A_2 \;R\; A_2'}{A_1\ \textsc{binary}\ A_2 \;\widehat{R}\; A_1'\ \textsc{binary}\ A_2'} \,.$$

And $\widehat{R}$ relates yielder terms with (arbitrary) matching subterms that are abstractions of actions related by R,

$$\frac{}{\textsf{given data} \;\widehat{R}\; \textsf{given data}} \qquad \frac{Y = \textsf{abstraction of}\ A \,,\quad A \;\; R \;\; A' \,,\quad \textsf{abstraction of}\ A' = Y'}{Y \;\; \widehat{R} \;\; Y'}$$

$$\frac{}{\textsf{current bindings} \;\widehat{R}\; \textsf{current bindings}} \qquad \frac{Y_1 \;\; \widehat{R} \;\; Y_1' \,,\ \ldots\ ,\ Y_n \;\; \widehat{R} \;\; Y_n' \,,\quad n \geqslant 0}{\textsc{data-op}_n(Y_1 \ldots Y_n) \;\; \widehat{R} \;\; \textsc{data-op}_n(Y_1' \ldots Y_n')} \,.$$

A relation R on actions is *compatible* on actions if it respects all action constructs. This can be expressed succinctly in terms of compatible refinement as $\widehat{R} \subseteq R$. It follows that every compatible relation R is reflexive. If R is also transitive, hence a preorder, we call R a *precongruence*.

The restriction of $\widehat{R}$ to data terms is *compatible* on data in the sense that for all $\textsc{data-op}_n$, $\textsc{data-op}_n(d_1 \ldots d_n) \;\widehat{R}\; \textsc{data-op}_n(d_1' \ldots d_n')$ whenever $d_1 \,\widehat{R}\, d_1' \ldots d_n \,\widehat{R}\, d_n'$.

5.2. *Uniformity*

Now we are in a position to formalise the uniformity requirement of data operations with respect to abstractions. We adopt an approach akin to relational parametricity (Reynolds 1983). Recall the grammar (1) for data terms from Assumption 2 in Section 3.6.4,

$$d \quad ::= \quad \textsf{abstraction of}\ A \quad | \quad \textsc{data-op}_n(d_1 \ldots d_n) \,.$$

Requirement 5.1. For every compatible relation R on actions and all data terms d_1, d_2, if $d_1 \;\widehat{R}\; d_2$,

1 whenever $d_1 = d_1'$ with d_1' generated by the above grammar, there exists $d_2' = d_2$ such that $d_1' \;\widehat{R}\; d_2'$, and
2 $d_1 : \textsf{data}$ if and only if $d_2 : \textsf{data}$.

Note that $d = d'$ means that d, d' are equated by the (initial) algebraic theory for sort data.

This is a property that all defining equations for data operations must respect so that all data rewriting is uniform with respect to abstractions. The requirement can also be defined by more detailed discussion of data terms and term contexts. However, our relational approach links up well with the semantic theory for actions developed below.

Let us see how the uniformity requirement excludes the examples from Section 3.6.4. For "is-and", choose any compatible relation R that relates $(A_1\ \textsf{or}\ A_2) \;\; R \;\; (A_1\ \textsf{and}\ A_2)$, for some actions A_1, A_2. Then

$$(\textsf{is-and abstraction of}\ (A_1\ \textsf{or}\ A_2)) \;\; \widehat{R} \;\; (\textsf{is-and abstraction of}\ (A_1\ \textsf{and}\ A_2)) \,,$$

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

but the left-hand side rewrites to false and the right-hand side does not; and false is a 0-ary data operation and thus only related to itself by $\widehat{R}$. Therefore "is-and" violates Part 1 of Requirement 5.1. Similarly, R shows that "and-left" from Section 3.6.4 violates Part 2 of the uniformity requirement.

Our requirements on data enable us to work on data, yielders, and actions up to equality in the algebraic theory while retaining a handle on the term structure. We shall only talk about *values* (in the initial algebraic model) and not syntactic *terms*. When we write $x\,\widehat{R}\,y$, we mean there exist terms s and t denoting x and y such that $s\,\widehat{R}\,t$ by the definition of $\widehat{R}$ on terms above.

With this interpretation we can prove the following yielder-substitutivity property of $\widehat{R}$ (recall yielder evaluation consists of substituting info into yielders).

Fact 5.2. If $i\ \widehat{R}\ i'$, $Y\ \widehat{R}\ Y'$ and $i \vdash Y \Downarrow d$, there exists d' such that $i' \vdash Y' \Downarrow d'$ and $d\ \widehat{R}\ d'$.

($\widehat{R}$ is defined for environments i, i':info because info $\leqslant$ data.)

Proof. The proof is by induction on the derivation of $i \vdash Y \Downarrow d$. □

5.3. *Contextual preorders*

Actions are contextually equivalent if they are interchangeable in all action contexts without making a difference in the observable effect. As observable effects, we take the completion predicates $\downarrow_{\text{MAY}}$ and $\downarrow_{\text{MUST}}$. Contextual equivalence is also known as 'observational congruence' as it is the largest congruence that respects these observations. This definition is fairly robust with respect to the exact choice of observations. Contextual equivalence can also be viewed as a testing equivalence (DeNicola and Hennessy 1984).

A great variety of stronger equivalences have been studied in the literature, especially bisimulation relations for processes. But in our setting, we find it difficult to justify a stronger semantic equivalence that distinguishes contextually equivalent actions. It is difficult to fix one semantic equivalence for action semantics. Our definition of contextual equivalence decomposes into the MAY and MUST modalities so that we obtain a small, expressive collection of equivalences. They provide a flexible theory, which should cover a wide range of application areas.

First we define environment indexed observation preorders $\ll_{\text{MAY}}$ and $\ll_{\text{MUST}}$.

Definition 5.3. (Observation preorders) For unfold-closed actions A_1, A_2 and environment i,

$$i \vdash A_1 \ll_m A_2 \quad \stackrel{\text{def}}{\Leftrightarrow} \quad i \vdash A_1 \downarrow_m \quad \Rightarrow \quad i \vdash A_2 \downarrow_m , \quad \text{for } m \in \{\text{MAY}, \text{MUST}\}.$$

When i is the empty environment, $i = ((),\{\})$, we just write $A_1 \ll_m A_2$.

An *action context* C is an action with any number of holes, denoted by [], occurring anywhere (even inside abstractions) such that C becomes a syntactically well-formed action whenever an action A is filled into the holes, written $C[A]$. A preorder R on actions is a *precongruence* if and only if it is closed under all action contexts, that is, $C[A]\ R\ C[A']$ whenever $A\ R\ A'$.

We define two contextual preorders $\sqsubseteq_{\text{MAY}}$ and $\sqsubseteq_{\text{MUST}}$ on actions as the greatest precongruence relations included in $\ll_{\text{MAY}}$ and $\ll_{\text{MUST}}$ for unfold-closed actions.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

Definition 5.4. (Contextual preorders) For arbitrary actions A_1, A_2,

$$A \sqsubseteq_{\simeq m} A' \stackrel{\text{def}}{\Leftrightarrow} (\forall \text{ action contexts } C \mid C[A], C[A'] \text{ are unfold-closed}) \;\; C[A] \ll_m C[A'] \; .$$

As a rough intuition, $A \sqsubseteq_{\simeq \text{MAY}} B$ and $A \sqsubseteq_{\simeq \text{MUST}} B$, both mean '$A$ is more divergent than B', and both $B \sqsubseteq_{\simeq \text{MAY}} A$ and $A \sqsubseteq_{\simeq \text{MUST}} B$ mean 'A is less deterministic than B'. We define the 'convex' preorder $\sqsubseteq_{\simeq \text{CVX}}$ as the conjunction of the MAY and MUST preorders and let $\simeq$ denote the induced equivalence. The divergent action "diverge" is least in all three preorders, and the completely unpredictable action "chaos" is the greatest element in the MAY preorder and least in the MUST preorder.

With these three contextual equivalences and preorders, we can express a variety of properties of actions by various equational and inequational laws, as is shown in Section 6.4.

In Section 6 we develop simulation methods for reasoning about action equivalences. Before that, we shall now discuss context lemmas that characterise the contextual preorders by more tractable sets of contexts than all action contexts. They provide alternative methods for proving actions equivalent. (We do not go into detail with this material and we omit proofs. In Lassen (1995) we prove these results for a basic fragment of action notation. These proofs scale up when combined with ideas from Smith's CIU proof in Smith (1992).)

5.4. *Finite restriction*

Our first context lemma says that it suffices to consider finite action contexts and @-substitutions in order to preorder actions. An action context, F, is called *finite* if it is unfold-closed and no holes occur under "unfolding": in other words, hole filling and @-substitution commute,

$$F[A]@B = F[A@B] \; , \;\; \text{for all actions } A, B. \tag{3}$$

For example, the action context "A then []" is finite if A is unfold-closed, but "unfolding (A then [])" is not finite.

Proposition 5.5. (Finite restriction) For arbitrary actions A, A',

$$A \sqsubseteq_{\simeq m} A' \;\; \Leftrightarrow \;\; (\forall \text{ finite } F, \text{ unfold-closed } B) \;\; F[A@B] \ll_m F[A'@B] \; .$$

Proposition 5.5 reduces preorderings of arbitrary actions to preordering unfold-closed actions (since $A@B, A'@B$ are unfold-closed when B is) and for unfold-closed actions, $\sqsubseteq_{\simeq m}$ is the largest relation closed under finite contexts and contained in $\ll_m$. This gives a slightly more tractable characterisation of the contextual preorders. It is possible, albeit cumbersome, to reason about $\sqsubseteq_{\simeq m}$ based on finite restriction. This technique is needed to prove Proposition 5.5 and can also be applied to show a recursion induction rule for the "unfolding" combinator.

Proposition 5.6. (Recursion induction) $\dfrac{A \;@\; B \sqsubseteq_{\simeq m} B}{\text{unfolding } A \sqsubseteq_{\simeq m} B}$,

for all actions A, B, and $m \in \{\text{MAY}, \text{MUST}, \text{CVX}\}$.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

The dynamic scope of actions sometimes makes recursion induction difficult to apply. The following rule gives a useful syntactic criteria for when unfolding does not interfere with information flow and it is safe to propagate information inside "unfolding":

Proposition 5.7. For all actions A, data d and bindings b, if $D = \mathsf{give}\ d$, $B = \mathsf{produce}\ b$,

$$\frac{D\ \mathsf{then\ unfolding}\ A\ \simeq\ (D\ \mathsf{then}\ A)@(D\ \mathsf{then\ unfolding}\ A)}{D\ \mathsf{then\ unfolding}\ A\ \simeq\ \mathsf{unfolding}\ (D\ \mathsf{then}\ A)}\ ,$$

$$\frac{B\ \mathsf{hence\ unfolding}\ A\ \simeq\ (B\ \mathsf{hence}\ A)@(B\ \mathsf{hence\ unfolding}\ A)}{B\ \mathsf{hence\ unfolding}\ A\ \simeq\ \mathsf{unfolding}\ (B\ \mathsf{hence}\ A)}\ .$$

The $\supseteq_{\mathrm{CVX}}$ direction of each rule is direct by recursion induction. The proof of the reverse is by finite restriction, similar to the proof of recursion induction.

Because of unbounded nondeterminism, 'domain-theoretic' reasoning principles hold only for the MAY preorder, *e.g.* an ω-induction rule:

$$\frac{(\forall n \geqslant 0)\ \ \mathsf{unfolding}^{(n)}\ A \sqsubseteq_{\mathrm{MAY}} B}{\mathsf{unfolding}\ A \sqsubseteq_{\mathrm{MAY}} B}\ , \tag{4}$$

where $\mathsf{unfolding}^{(0)}\ A \stackrel{\mathrm{def}}{=} \mathsf{diverge}$, $\mathsf{unfolding}^{(n+1)}\ A \stackrel{\mathrm{def}}{=} A@\mathsf{unfolding}^{(n)}\ A$.

It is probably worthwhile to develop this kind of reasoning principle for settings where the infinitely branching primitive "choose" is not used, *e.g.* in the example action semantic description in Section 2. Nevertheless, we shall not venture to do so in the present paper (and it also seems that most often continuity arguments can be replaced by recursion induction or by the co-inductive techniques in Section 6).

5.5. *Experimental restriction*

Finite restriction reduces the contexts required to preorder actions to finite action contexts and @-substitutions. For unfold-closed actions there is an analogous reduction to evaluation contexts and environments. An *evaluation context*, E, is a finite context with one hole that occurs at redex position in the action. Evaluation contexts are the unfold-closed contexts given by the following grammar, where [] denotes the hole at redex position.

- evaluation-context = [] | furthermore evaluation-context |
  evaluation-context BINARY action |
  action or evaluation-context | action and evaluation-context.

Proposition 5.8. (Experimental restriction) For unfold-closed actions A, A',

$$A \sqsubseteq_m A' \quad \Leftrightarrow \quad (\forall\ \text{evaluation contexts}\ E,\ i : \mathsf{info})\ \ i \vdash E[A] \ll_m E[A']\ .$$

This corresponds to the CIU theorem of Mason and Talcott (1991) and Gordon's 'experimental' characterisation of contextual equivalence (Gordon 1995a).

We can use experimental restriction to prove the following syntactic representation of the quantification over environments in Proposition 5.8.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

Proposition 5.9. For unfold-closed A, A',

$$\begin{aligned} A \mathrel{\widetilde{\sqsubseteq}}_m A' &\Leftrightarrow (\forall\, d{:}\mathsf{data})\ \mathsf{give}\ d\ \mathsf{then}\ A \mathrel{\widetilde{\sqsubseteq}}_m \mathsf{give}\ d\ \mathsf{then}\ A' \\ &\Leftrightarrow (\forall\, b{:}\mathsf{bindings})\ \mathsf{produce}\ b\ \mathsf{hence}\ A \mathrel{\widetilde{\sqsubseteq}}_m \mathsf{produce}\ b\ \mathsf{hence}\ A' \ . \end{aligned}$$

The proof involves analysis of evaluation contexts and evaluation. Because of nondeterminism, this is more labourious than in sequential settings, such as Mason and Talcott's. In the present exposition we shall instead focus on the more informative simulation techniques presented in the next section.

6. Simulation

In this section we develop a simple and powerful simulation proof technique for establishing contextual equivalences and preorderings on actions. The soundness is established by proving that associated simulation preorders are precongruences. We also show that simulation is incomplete; nonetheless, simulation is very useful for reasoning about actions and we outline algebraic laws for actions that are easily proved by simulation.

6.1. *Simulation preorders*

We define simulation co-inductively, inspired by the bisimulation for actions in Mosses (1992, Chapter 4). Our MAY and MUST formulation takes inspiration from Ulidowski's copy+refusal testing for processes (Ulidowski 1992) and Ong's applicative bisimulation for nondeterministic lambda calculus (Ong 1993).

Let the MAY simulation preorder be the greatest relation, $\lesssim_{\text{MAY}}$, on unfold-closed actions satisfying

Whenever $A \lesssim_{\text{MAY}} A'$ and $i \vdash A \Downarrow^{\text{MAY}} d{\times}b$, also $i \vdash A' \Downarrow^{\text{MAY}} d'{\times}b'$ with $d{\times}b \widetilde{\lesssim}_{\text{MAY}} d'{\times}b'$.

And the MUST simulation preorder is the greatest relation, $\lesssim_{\text{MUST}}$, on unfold-closed actions satisfying

Whenever $A \lesssim_{\text{MUST}} A'$ and $i \vdash A \Downarrow^{\text{MUST}}$, also $i \vdash A' \Downarrow^{\text{MUST}}$ and for all t' if $i \vdash A' \Downarrow^{\text{MAY}} t'$ then $i \vdash A \Downarrow^{\text{MAY}} t$ for some $t \widetilde{\lesssim}_{\text{MUST}} t'$.

For all relations R on actions, $\widetilde{R}$ relates data terms that are syntactically identical except for subterms that are abstractions of actions pairwise related by R.

$$\frac{d = \mathsf{abstraction\ of}\ A\ , \quad A \ \ R \ \ A'\ , \quad \mathsf{abstraction\ of}\ A' = d'}{d \ \ \widetilde{R} \ \ d'}$$

$$\frac{d_1 \ \ \widetilde{R} \ \ d_1'\ ,\ \ldots\ ,\ d_n \ \ \widetilde{R}\ d_n'\ , \quad n \geqslant 0}{\text{DATA-OP}_n(d_1 \ldots d_n) \ \ \widetilde{R} \ \ \text{DATA-OP}_n(d_1' \ldots d_n')} \ \textit{if not}\ \text{DATA-OP}_n(d_1 \ldots d_n){:}\mathsf{abstraction}.$$

We extend $\widetilde{R}$ to relate outcomes also,

$$\frac{}{\mathsf{failed} \ \ \widetilde{R} \ \ \mathsf{failed}} \qquad \frac{d \ \widetilde{R}\ d', \quad b \ \widetilde{R}\ b'}{d{\times}b \ \ \widetilde{R} \ \ d'{\times}b'} \ .$$

On data terms, $\widetilde{R}$ is defined almost like compatible refinement but $\widetilde{R}$ is only closed under

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

data operations with co-domain other than abstraction. For compatible relations they coincide.

Fact 6.1. If R is compatible, $d\ \widehat{R}\ d'$ iff $d\ \widetilde{R}\ d'$, for all data terms d, d'.

Contrary to compatible refinement, the $\widetilde{\ }$ operation commutes with relation composition.

Fact 6.2. $\widetilde{R}\,\widetilde{S} = \widetilde{R\,S}$, for all relations R, S on actions.

(We write relation composition by juxtapositioning, $x\ R\,S\ y \stackrel{\text{def}}{\Leftrightarrow} (\exists z)\ x\ R\ z\ \wedge\ z\ S\ y$.)

We can express the simulation preorders in terms of simulation operators $\langle_\rangle_{\text{MAY}}$, $\langle_\rangle_{\text{MUST}}$.

Definition 6.3. For every relation R on actions, $\langle R\rangle_m$ relates unfold-closed actions A, A' as follows:

$$A\langle R\rangle_{\text{MAY}}\ A' \stackrel{\text{def}}{\Leftrightarrow} (\forall i)\,(\forall\, c : \mathsf{completed} \mid i \vdash A \Downarrow^{\text{MAY}} c)\ (\exists\, c' : \mathsf{completed} \mid i \vdash A' \Downarrow^{\text{MAY}} c')\ \ c\,\widetilde{R}\,c'\ .$$

$$A\langle R\rangle_{\text{MUST}}\ A' \stackrel{\text{def}}{\Leftrightarrow} (\forall i \mid i \vdash A \Downarrow^{\text{MUST}})\ \ i \vdash A' \Downarrow^{\text{MUST}}\ \wedge\ (\forall t' \mid i \vdash A' \Downarrow^{\text{MAY}} t')\,(\exists t \mid i \vdash A \Downarrow^{\text{MAY}} t)\ \ t\,\widetilde{R}\,t'\ .$$

Both $\langle_\rangle_m$ are monotone operators on the complete lattice of relations R on unfold-closed actions, ordered by subset inclusion. By Tarski's theorem, each $\langle_\rangle_m$ has a greatest fixpoint, namely $\lesssim_m$.

Co-induction on greatest fixpoints gives a simulation proof rule for the simulation preorders.

Rule 6.4. (Simulation) $\dfrac{R \subseteq \langle R\rangle_m}{R \subseteq \lesssim_m}$.

The premiss of the simulation rules requires R to be a post-fixpoint of $\langle_\rangle_m$. Notice that many problems concerning semantic preorders or equivalences have the format of the conclusion of the rule – R may be a singleton relation relating two actions to be shown preordered, or R may contain all instances of a general action law. For example, the simulation preorders are reflexive and transitive (so they are indeed preorders), because the identity relation and $\lesssim_m \lesssim_m$ are post-fixpoints of $\langle_\rangle_m$. (The latter relies on Fact 6.2.)

Many results follow just from the facts that $\lesssim_m$ is a fixpoint, $\lesssim_m\ =\ \langle \lesssim_m \rangle_m$ and that $\lesssim_m$ and $\widetilde{\lesssim_m}$ are reflexive. For example,

$$\mathsf{choose\ natural}\ \sim_{\text{MAY}}\ \mathsf{generate}\ ,$$

where $\sim_m$ denotes the symmetric closure of $\lesssim_m$, $\sim_m \stackrel{\text{def}}{=} \lesssim_m \cap \gtrsim_m$. They are MAY simulation equivalent, because both evaluate to the same set of outcomes; but the equivalence fails for MUST, because only "choose natural" must terminate (*cf.* Section 4).

As an example of co-induction on infinite data structures, consider the following simulation equivalence: for $m \in \{\text{MAY}, \text{MUST}\}$,

$$\mathsf{evaluate}\ [\![\mathbf{rec}\ x.\lambda y.x]\!]\ \sim_m\ B \stackrel{\text{def}}{=} \mathsf{unfolding\ give\ abstraction\ of\ unfold}\ . \qquad (5)$$

rec $x.\lambda y.x$ is a function that, regardless of its input, returns itself. B gives an abstraction that, when enacted, again gives the same abstraction. First we observe, for all $(d,\ b)$, i :

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

info,

$$(d,\ b) \vdash \text{evaluate } [\![\textbf{rec } x.\lambda y.x]\!] \Downarrow^{\text{MAY}} (\text{abstraction of } A_{(b)})\times\{\}\ ,$$
$$i \vdash A_{(b)} \Downarrow^{\text{MAY}} (\text{abstraction of } A_{(b)})\times\{\}\ ,$$
$$i \vdash B \Downarrow^{\text{MAY}} (\text{abstraction of } B)\times\{\}\ .$$

where $A_{(b)} = \text{produce overlay}(\{x \mapsto a\},\ b)$ hence (furthermore bind y to given data hence enact the data bound to x) ;

$a =$ abstraction of (produce b hence evaluate $[\![\textbf{rec } x.\lambda y.x]\!]$) .

Let $R \stackrel{\text{def}}{=} \{(A,B),(A_{(b)},B),(B,A),(B,A_{(b)}) \mid b\text{:bindings}\}$. Now it is immediate from the definition of $\langle_\rangle_m$ that $R \subseteq \langle R\rangle_m$, hence $R \subseteq \lesssim_m$ and, by symmetry, $R \subseteq \sim_m$, and thus (5) holds.

6.2. *Precongruence*

In order to establish our main result, that simulation is sound with respect to the contextual preorders, we first show that the simulation preorders are precongruences. We apply Howe's general method (Howe 1989), which has become a standard approach in connection with (bi)simulation for higher-order languages. We borrow ideas from Ong's application of Howe's method to a mixed call-by-value/call-by-name nondeterministic lambda calculus (Ong 1992). Ong's applicative bisimulation resembles our MAY and MUST simulations. We extend this work to cover unbounded nondeterminism, dynamic bindings and abstract data types.

The simulation preorders are defined only for unfold-closed actions. First we extend them to relations on arbitrary actions, $\lesssim_m^\circ$, defined in terms of closure under @-substitutions,

$$A \lesssim_m^\circ A' \quad \stackrel{\text{def}}{\Leftrightarrow} \quad (\forall \text{ unfold-closed } B)\ \ A@B \lesssim_m A'@B\ .$$

Howe's method works as follows. For $m \in \{\text{MAY}, \text{MUST}\}$, we define a relation $\lesssim_m^\bullet$ that includes $\lesssim_m^\circ$ and is compatible, $\widehat{\lesssim_m^\bullet} \subseteq \lesssim_m^\bullet$. Then we show $\lesssim_m^\bullet$ is included in $\lesssim_m^\circ$ by simulation. Thus $\lesssim_m^\circ$ and $\lesssim_m^\bullet$ coincide, and we get $\widehat{\lesssim_m^\circ} \subseteq \lesssim_m^\circ$, and thus $\lesssim_m^\circ$ is a precongruence.

Definition 6.5. $\lesssim_m^\bullet$ is the least relation on actions such that $\lesssim_m^\bullet = \widehat{\lesssim_m^\bullet}\lesssim_m^\circ$.

(Compatible refinement and relation composition are monotone, so a least solution exists.)

Lemma 6.6. $\lesssim_m^\bullet$ is reflexive, (*i*) $\widehat{\lesssim_m^\bullet} \subseteq \lesssim_m^\bullet$, (*ii*) $\lesssim_m^\circ \subseteq \lesssim_m^\bullet$, (*iii*) $\lesssim_m^\bullet \lesssim_m^\circ \subseteq \lesssim_m^\bullet$.

Proof. (*i*) holds by definition of $\lesssim_m^\bullet$ because $\lesssim_m^\circ$ is reflexive. So $\lesssim_m^\bullet$ is compatible and therefore also reflexive and (*ii*) follows by the same argument as for the first. We obtain (*iii*) from the definition of $\lesssim_m^\bullet$ and transitivity of $\lesssim_m^\circ$. □

$\lesssim_m^\bullet$ satisfies the following '@-substitutivity' and 'yielder-substitutivity' properties:

Lemma 6.7. For all actions A, A', B, B' and yielders Y, Y',
if $A \lesssim_m^\bullet A'$, $Y \widehat{\lesssim_m^\bullet} Y'$, and $B \lesssim_m^\bullet B'$, then $A@B \lesssim_m^\bullet A'@B'$ and $Y@B \widehat{\lesssim_m^\bullet} Y'@B'$.

Proof. The proof is by simultaneous induction on the derivation of $A \lesssim_m^\bullet A'$ and $Y \widehat{\lesssim_m^\bullet} Y'$. □

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

Lemma 6.8. If $i \widetilde{\lesssim^{\bullet}_{m}} i'$, $Y \widehat{\lesssim^{\bullet}_{m}} Y'$, and $i \vdash Y \Downarrow d$, there exists d' such that $i' \vdash Y' \Downarrow d'$ and $d \widetilde{\lesssim^{\bullet}_{m}} d'$.

Proof. The proof follows from Fact 6.1 and Fact 5.2. □

We will now show $\lesssim^{\bullet}_{m} \subseteq \lesssim^{\circ}_{m}$ by means of simulation. In the MAY case, the proof is by induction on action evaluation, $\Downarrow^{\text{MAY}}$.

Lemma 6.9. $\lesssim^{\bullet}_{\text{MAY}} \subseteq \lesssim^{\circ}_{\text{MAY}}$.

Proof. Since $\lesssim^{\bullet}_{\text{MAY}}$ is reflexive and substitutive, it suffices to show $A \lesssim^{\bullet}_{\text{MAY}} A'$ implies $A \lesssim_{\text{MAY}} A'$, for all unfold-closed A, A'. This follows by simulation if

$$(\forall \text{ unfold-closed } A, A' \mid A \lesssim^{\bullet}_{\text{MAY}} A') \;\; A \left\langle \lesssim^{\bullet}_{\text{MAY}} \right\rangle_{\text{MAY}} A' \,. \tag{6}$$

We prove a slightly stronger claim:

$$\begin{aligned} &(\forall i)\,(\forall \text{ unfold-closed } A)\,(\forall c : \mathsf{completed} \mid i \vdash A \Downarrow^{\text{MAY}} c) \\ &\quad (\forall i')\,(\forall \text{ unfold-closed } A' \mid A \lesssim^{\bullet}_{\text{MAY}} A' \;\wedge\; i \widetilde{\lesssim^{\bullet}_{\text{MAY}}} i') \\ &\qquad (\exists c' : \mathsf{completed} \mid i' \vdash A' \Downarrow^{\text{MAY}} c') \;\; c \widetilde{\lesssim^{\bullet}_{\text{MAY}}} c' \,. \end{aligned} \tag{7}$$

(This entails (6) by taking $i' = i$.)

The proof is by induction on the derivation of $i \vdash A \Downarrow^{\text{MAY}} c$. Assume $A \lesssim^{\bullet}_{\text{MAY}} A'$ and $i \widetilde{\lesssim^{\bullet}_{\text{MAY}}} i'$. By definition of $\lesssim^{\bullet}_{\text{MAY}}$, there exists $A'' \lesssim_{\text{MAY}} A'$ such that $A \widehat{\lesssim^{\bullet}_{\text{MAY}}} A''$. It suffices to show

$$(\exists c'' \mid i' \vdash A'' \Downarrow^{\text{MAY}} c'') \;\; c \widetilde{\lesssim^{\bullet}_{\text{MAY}}} c'' \,, \tag{8}$$

because then $A'' \lesssim_{\text{MAY}} A'$ implies $i' \vdash A' \Downarrow^{\text{MAY}} c'$ with $c'' \widetilde{\lesssim_{\text{MAY}}} c'$, and we can conclude $c \widetilde{\lesssim^{\bullet}_{\text{MAY}}} c'$ because $\widetilde{\lesssim^{\bullet}_{\text{MAY}}}\, \widetilde{\lesssim_{\text{MAY}}} \subseteq \widetilde{\lesssim^{\bullet}_{\text{MAY}}}\, \widetilde{\lesssim_{\text{MAY}}} \subseteq \widetilde{\lesssim^{\bullet}_{\text{MAY}}}$ by Fact 6.2 and Lemma 6.6(*iii*). We proceed by analysis of the derivation of $i \vdash A \Downarrow^{\text{MAY}} c$. Some illustrative evaluation rules are (*cf.* Section 4.2)

$\Downarrow^{\text{MAY}}(2)$ $A = \mathsf{enact}\ Y$; $i \vdash Y \Downarrow d$; $d = \mathsf{abstraction\ of}\ B$; $((),\{\}) \vdash B \Downarrow^{\text{MAY}} c$.

$A \widehat{\lesssim^{\bullet}_{\text{MAY}}} A'' = \mathsf{enact}\ Y''$ with $Y \widehat{\lesssim^{\bullet}_{\text{MAY}}} Y''$ and, by Lemma 6.8, $i' \vdash Y'' \Downarrow d''$ with $d \widetilde{\lesssim^{\bullet}_{\text{MAY}}} d''$, hence $d'' = \mathsf{abstraction\ of}\ B''$ with $B \lesssim^{\bullet}_{\text{MAY}} B''$. $((),\{\}) \widetilde{\lesssim^{\bullet}_{\text{MAY}}} ((),\{\})$ holds because $\lesssim^{\bullet}_{\text{MAY}}$ and $\widetilde{\lesssim^{\bullet}_{\text{MAY}}}$ are reflexive. So, by the induction hypothesis, we have $((),\{\}) \vdash B'' \Downarrow^{\text{MAY}} c''$ with $c \widetilde{\lesssim^{\bullet}_{\text{MAY}}} c''$, and we deduce $i' \vdash A'' \Downarrow^{\text{MAY}} c''$, as required.

$\Downarrow^{\text{MAY}}(6)$ $A = A_1\ \mathsf{or}\ A_2$; without loss of generality, assume $i \vdash A_1 \Downarrow^{\text{MAY}} c$.

$A \widehat{\lesssim^{\bullet}_{\text{MAY}}} A'' = (A_1'\ \mathsf{or}\ A_2')$ with $A_1 \lesssim^{\bullet}_{\text{MAY}} A_1'$, $A_2 \lesssim^{\bullet}_{\text{MAY}} A_2'$. By the induction hypothesis, $i' \vdash A_1' \Downarrow^{\text{MAY}} c''$ with $c \widetilde{\lesssim^{\bullet}_{\text{MAY}}} c''$, hence also $i' \vdash A'' \Downarrow^{\text{MAY}} c''$ by evaluation rule $\Downarrow^{\text{MAY}}(6)$.

$\Downarrow^{\text{MAY}}(9)$ $A = A_1\ \mathsf{then}\ A_2$; $i = (d, b)$; $i \vdash A_1 \Downarrow^{\text{MAY}} d_1 \times b_1$; $(d_1, b) \vdash A_2 \Downarrow^{\text{MAY}} d_2 \times b_2$; $c = d_2 \times \mathsf{merge}(b_1, b_2)$.

$A \widehat{\lesssim^{\bullet}_{\text{MAY}}} A'' = (A_1'\ \mathsf{then}\ A_2')$ with $A_1 \lesssim^{\bullet}_{\text{MAY}} A_1'$, $A_2 \lesssim^{\bullet}_{\text{MAY}} A_2'$; $i \widetilde{\lesssim^{\bullet}_{\text{MAY}}} i' = (d', b')$ with $d \widetilde{\lesssim^{\bullet}_{\text{MAY}}} d'$, $b \widetilde{\lesssim^{\bullet}_{\text{MAY}}} b'$. By the induction hypothesis, $i' \vdash A_1' \Downarrow^{\text{MAY}} d_1' \times b_1'$ with $d_1 \widetilde{\lesssim^{\bullet}_{\text{MAY}}} d_1'$, $b_1 \widetilde{\lesssim^{\bullet}_{\text{MAY}}} b_1'$. So $(d_1, b) \widetilde{\lesssim^{\bullet}_{\text{MAY}}} (d_1', b')$ and by the induction hypothesis, $(d_1', b') \vdash A_2' \Downarrow^{\text{MAY}} d_2' \times b_2'$ with $d_2 \widetilde{\lesssim^{\bullet}_{\text{MAY}}} d_2'$, $b_2 \widetilde{\lesssim^{\bullet}_{\text{MAY}}} b_2'$, and hence we have $c \widetilde{\lesssim^{\bullet}_{\text{MAY}}} c'' = d_2' \times \mathsf{merge}(b_1', b_2')$ and $i' \vdash A'' \Downarrow^{\text{MAY}} c''$.

$\Downarrow^{\text{MAY}}(12)$ $A = \mathsf{unfolding}\ A_1$; $i \vdash A_1 @ A \Downarrow^{\text{MAY}} c$.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

$A \mathrel{\widehat{\lesssim^{\bullet}_{\mathrm{MAY}}}} A'' = (\mathsf{unfolding}\ A_1')$ with $A_1 \lesssim^{\bullet}_{\mathrm{MAY}} A_1'$. So $(A_1 @ A) \lesssim^{\bullet}_{\mathrm{MAY}} (A_1' @ A'')$ by compatibility and @-substitutivity of $\lesssim^{\bullet}_{\mathrm{MAY}}$, Lemma 6.6(*i*) and Lemma 6.7. By the induction hypothesis, $i' \vdash A_1' @ A'' \Downarrow^{\mathrm{MAY}} c''$ with $c \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MAY}}}} c''$, hence also $i' \vdash A'' \Downarrow^{\mathrm{MAY}} c''$.

The remaining cases are similar or simpler, and we conclude (8), as required. □

In the MUST case, the proof is based on the MUST part of Ong's congruence proof for applicative bisimulation in (Ong 1992). Ong's proof is by mathematical induction on the length of computation whereas our argument is phrased more abstractly in terms of induction on the derivation of must termination, $\Downarrow^{\mathrm{MUST}}$. In particular, we do not count reduction steps, and therefore unbounded nondeterminism causes no problems and we eschew explicit transfinite induction.

Lemma 6.10. $\lesssim^{\bullet}_{\mathrm{MUST}} \subseteq \lesssim^{\circ}_{\mathrm{MUST}}$.

Proof. As for the MAY case above, we argue by simulation and show

$$(\forall \text{ unfold-closed } A, A' \mid A \lesssim^{\bullet}_{\mathrm{MUST}} A') \quad A \left\langle \lesssim^{\bullet}_{\mathrm{MUST}} \right\rangle_{\mathrm{MUST}} A' \ . \tag{9}$$

This follows if

$$\begin{aligned} &(\forall i)\,(\forall \text{ unfold-closed } A \mid i \vdash A \Downarrow^{\mathrm{MUST}}) \\ &\quad (\forall i')\,(\forall \text{ unfold-closed } A' \mid A \lesssim^{\bullet}_{\mathrm{MUST}} A' \ \wedge\ i \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} i') \\ &\qquad\qquad i' \vdash A' \Downarrow^{\mathrm{MUST}} \quad \wedge \quad (\forall t' \mid i' \vdash A' \Downarrow^{\mathrm{MAY}} t')\,(\exists t \mid i \vdash A \Downarrow^{\mathrm{MAY}} t) \ \ t \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} t' \ . \end{aligned} \tag{10}$$

We prove (10) by induction on the derivation of $i \vdash A \Downarrow^{\mathrm{MUST}}$. If $A \lesssim^{\bullet}_{\mathrm{MUST}} A'$ and $i \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} i'$, there exists $A'' \lesssim_{\mathrm{MUST}} A'$ such that $A \mathrel{\widehat{\lesssim^{\bullet}_{\mathrm{MUST}}}} A''$. We will show

$$\begin{aligned} &\text{(I)} \quad i' \vdash A'' \Downarrow^{\mathrm{MUST}} \quad \wedge \\ &\text{(II)} \quad (\forall t'' \mid i' \vdash A'' \Downarrow^{\mathrm{MAY}} t'')\,(\exists t \mid i \vdash A \Downarrow^{\mathrm{MAY}} t) \ \ t \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} t'' \ , \end{aligned} \tag{11}$$

then (10) follows by definition of $\lesssim_{\mathrm{MUST}}$, Fact 6.2 and Lemma 6.6(*iii*). We proceed by analysis of the derivation of $i \vdash A \Downarrow^{\mathrm{MUST}}$. We discuss a few termination rules (*cf.* Section 4.3):

$\Downarrow^{\mathrm{MUST}}(2)$ $A = \mathsf{enact}\ Y \ ;\ i \vdash Y \Downarrow d \ ;\ d = \mathsf{abstraction\ of}\ B \ ;\ ((),\{\}) \vdash B \Downarrow^{\mathrm{MUST}}$.

$A \mathrel{\widehat{\lesssim^{\bullet}_{\mathrm{MUST}}}} A'' = \mathsf{enact}\ Y''$ with $Y \mathrel{\widehat{\lesssim^{\bullet}_{\mathrm{MUST}}}} Y''$. By Lemma 6.8, $i' \vdash Y'' \Downarrow d''$ with $d \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} d''$, hence $d'' = \mathsf{abstraction\ of}\ B''$ with $B \lesssim^{\bullet}_{\mathrm{MUST}} B''$. Now observe that $((),\{\}) \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} ((),\{\})$.

(I) By the induction hypothesis, $((),\{\}) \vdash B'' \Downarrow^{\mathrm{MUST}}$, and hence $i' \vdash A'' \Downarrow^{\mathrm{MUST}}$.

(II) If $i' \vdash A'' \Downarrow^{\mathrm{MAY}} t''$, also $((),\{\}) \vdash B'' \Downarrow^{\mathrm{MAY}} t''$ (since yielder evaluation is deterministic) and again by the induction hypothesis, $((),\{\}) \vdash B \Downarrow^{\mathrm{MAY}} t$ with $t \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} t''$, hence also $i \vdash A \Downarrow^{\mathrm{MAY}} t$ as required.

$\Downarrow^{\mathrm{MUST}}(6)$ $A = A_1\ \mathsf{or}\ A_2 \ ;\ i \vdash A_1 \Downarrow^{\mathrm{MUST}} \ ;\ i \vdash A_2 \Downarrow^{\mathrm{MUST}}$.

$A \mathrel{\widehat{\lesssim^{\bullet}_{\mathrm{MUST}}}} A'' = (A_1'\ \mathsf{or}\ A_2')$ with $A_1 \lesssim^{\bullet}_{\mathrm{MUST}} A_1'$, $A_2 \lesssim^{\bullet}_{\mathrm{MUST}} A_2'$.

(I) By the induction hypothesis, $i' \vdash A_1' \Downarrow^{\mathrm{MUST}}$ and $i' \vdash A_2' \Downarrow^{\mathrm{MUST}}$, and hence we have $i' \vdash A'' \Downarrow^{\mathrm{MUST}}$.

(II) Suppose $i' \vdash A'' \Downarrow^{\mathrm{MAY}} t''$. If $t'' : \mathsf{completed}$, $i' \vdash A_1' \Downarrow^{\mathrm{MAY}} t''$ or $i' \vdash A_2' \Downarrow^{\mathrm{MAY}} t''$; without loss of generality, assume the former and by the induction hypothesis conclude $i \vdash A_1 \Downarrow^{\mathrm{MAY}} t$ with $t \mathrel{\widetilde{\lesssim^{\bullet}_{\mathrm{MUST}}}} t''$ and thus $t : \mathsf{completed}$, hence also $i \vdash A \Downarrow^{\mathrm{MAY}} t$. If

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

$t'' = \mathsf{failed}$, both $i' \vdash A'_1 \Downarrow^{\text{MAY}} \mathsf{failed}$ and $i' \vdash A'_2 \Downarrow^{\text{MAY}} \mathsf{failed}$, and by the induction hypothesis it follows that $i \vdash A_1 \Downarrow^{\text{MAY}} \mathsf{failed}$ and $i \vdash A_2 \Downarrow^{\text{MAY}} \mathsf{failed}$, hence also $i \vdash A \Downarrow^{\text{MAY}} \mathsf{failed}$.

$\Downarrow^{\text{MUST}}(9)$ $A = A_1 \ \mathsf{then} \ A_2 \ ; \ i = (d, b) \ ; \ i \vdash A_1 \Downarrow^{\text{MUST}}; \ (\forall d_1, b_1 \mid i \vdash A_1 \Downarrow^{\text{MAY}} d_1 \times b_1)$ $(d_1, b) \vdash A_2 \Downarrow^{\text{MUST}}$.
$A \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ A'' = (A'_1 \ \mathsf{then} \ A'_2)$ with $A_1 \lesssim^{\bullet}_{\text{MUST}} A'_1, A_2 \lesssim^{\bullet}_{\text{MUST}} A'_2 \ ; \ i \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ i' = (d', b')$ with $d \widetilde{\lesssim^{\bullet}_{\text{MUST}}} d', b \widetilde{\lesssim^{\bullet}_{\text{MUST}}} b'$.

(I) First we show $i' \vdash A'' \Downarrow^{\text{MUST}}$. By the induction hypothesis, $i' \vdash A'_1 \Downarrow^{\text{MUST}}$ and whenever $i' \vdash A'_1 \Downarrow^{\text{MAY}} d'_1 \times b'_1$, $i \vdash A_1 \Downarrow^{\text{MAY}} d_1 \times b_1$ with $d_1 \times b_1 \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ d'_1 \times b'_1$, hence $(d_1, b) \vdash A_2 \Downarrow^{\text{MUST}}$, $(d_1, b) \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ (d'_1, b')$ and, by the induction hypothesis, we have $(d'_1, b') \vdash A'_2 \Downarrow^{\text{MUST}}$, and we conclude $i' \vdash A'' \Downarrow^{\text{MUST}}$.

(II) Next, suppose $i' \vdash A'' \Downarrow^{\text{MAY}} t''$. This can be derived from $i' \vdash A'_1 \Downarrow^{\text{MAY}} d'_1 \times b'_1$ and $(d'_1, b') \vdash A'_2 \Downarrow^{\text{MAY}} t'_2$ and either $t'_2 = d'_2 \times b'_2$, $t'' = d'_2 \times \mathsf{merge}(b'_1, b'_2)$ or $t'_2 = t'' = \mathsf{failed}$. By the induction hypothesis, $i \vdash A_1 \Downarrow^{\text{MAY}} d_1 \times b_1$ with $d_1 \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ d'_1$, $b_1 \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ b'_1$. Hence $(d_1, b) \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ (d'_1, b')$ and, again we have by the induction hypothesis, $(d_1, b) \vdash A_2 \Downarrow^{\text{MAY}} t_2$ with $t_2 \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ t'_2$, and we can compute $t \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ t''$ such that $i \vdash A \Downarrow^{\text{MAY}} t$. If $t'' = \mathsf{failed}$, $i' \vdash A'' \Downarrow^{\text{MAY}} t''$ can also be derived from $i' \vdash A'_1 \Downarrow^{\text{MAY}} \mathsf{failed}$, but then $i \vdash A_1 \Downarrow^{\text{MAY}} \mathsf{failed}$ is immediate from the induction hypothesis for $i \vdash A_1 \Downarrow^{\text{MUST}}$, and hence $i \vdash A \Downarrow^{\text{MAY}} \mathsf{failed}$, as required.

$\Downarrow^{\text{MUST}}(12)$ $A = \mathsf{unfolding} \ A_1 \ ; \ i \vdash A_1 @ A \Downarrow^{\text{MUST}}$.
$A \ \widetilde{\lesssim^{\bullet}_{\text{MUST}}} \ A'' = (\mathsf{unfolding} \ A'_1)$ with $A_1 \lesssim^{\bullet}_{\text{MUST}} A'_1$. So $(A_1 @ A) \lesssim^{\bullet}_{\text{MUST}} (A'_1 @ A'')$, by compatibility and @-substitutivity of $\lesssim^{\bullet}_{\text{MUST}}$, and the result follows from the induction hypothesis.

The remaining cases are similar or simpler and we conclude (11) as required. □

Theorem 6.11. $\lesssim^{\circ}_{m}$ is a precongruence.

Proof. The proof follows from Lemma 6.6(*i*)–(*ii*), 6.9, and 6.10. □

6.3. *Soundness and incompleteness*

Given that the simulation preorders are precongruences, it easily follows that they are sound approximations of the contextual preorders.

Theorem 6.12. (Soundness) $\lesssim^{\circ}_{m} \subseteq \sqsubseteq_{m}$.

Proof. By definition, $\sqsubseteq_{m}$ is the largest precongruence contained in $\ll_{m}$ for unfold-closed actions. Since $\lesssim^{\circ}_{m}$ is a precongruence, we only need to show $\lesssim_{m} \subseteq \ll_{m}$. For MAY, the result is immediate from the definition of $\lesssim_{\text{MAY}}$. For MUST, if $A \lesssim_{\text{MUST}} A'$ and $i \vdash A \downarrow_{\text{MUST}}$, then also $i \vdash A \Downarrow^{\text{MUST}}$, $i \vdash A' \Downarrow^{\text{MUST}}$, and whenever $i \vdash A' \Downarrow^{\text{MAY}} t'$, also $i \vdash A \Downarrow^{\text{MAY}} t$ for some $t \ \widetilde{\lesssim_{\text{MUST}}} \ t'$. But $t : \mathsf{completed}$ because $i \vdash A \downarrow_{\text{MUST}}$, therefore also $t' : \mathsf{completed}$. Consequently, $i \vdash A' \downarrow_{\text{MUST}}$ and we conclude $\lesssim_{\text{MUST}} \subseteq \ll_{\text{MUST}}$. □

This result is important as it enables us to reason about the contextual preorders and equivalence by simulation. Conversely, we may ask whether the simulation orders are also

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

complete: do they coincide with the contextual preorders? The answer is negative because the simulation preorders take branching structure of nondeterminism into account, which the contextual preorders abstract from.

The following example shows that the MAY simulation preorder is not complete. We discuss the proof in some detail as it illustrates how to use simulation and induction rules to reason about actions.

Proposition 6.13. give abstraction of generate $\sqsubseteq_{\text{MAY}}$
generate then give application of (abstraction of A) to given data ,
where A = (generate and regive) then give min(first of given data, rest of given data) .
("min" gives the minimum of two natural numbers and is defined in Appendix A.)

Proof. First we use recursion induction to obtain

$$\text{give abstraction of generate} \simeq \text{unfolding } B \,, \tag{12}$$

where B = give abstraction of (give 0 or (unfold then enact given data then
give successor of given data)) .
The $\sqsupseteq_{\text{CVX}}$ direction is straightforward. The reverse is done by unfolding the right-hand side and proving

$$\text{generate} \sqsubseteq_{\text{CVX}} \text{give 0 or (unfolding } B \text{ then enact given data then give successor of given data)} \,,$$

by recursion induction.

Next we apply the ω-induction rule for the MAY contextual preorder, (4), to prove

$$\text{unfolding } B \sqsubseteq_{\text{MAY}} \text{generate then give application of (abstraction of } A\text{) to given data} \,. \tag{13}$$

We show

$$(\forall n \geqslant 0)\ \ \text{unfolding}^{(n)}\ B \lesssim_{\text{MAY}} \text{give abstraction of (give } n \text{ then } A) \,, \tag{14}$$

by induction on n. The base case is immediate. For the induction step, observe

$$i \vdash \text{unfolding}^{(n+1)}\ B \Downarrow^{\text{MAY}} \text{abstraction of } B' \,,$$

where B' = give 0 or (unfolding$^{(n)}$ B then enact given data then
give successor of given data).
And by the induction hypothesis and simulation we get $B' \lesssim_{\text{MAY}}$ give n then A and conclude (14) holds for $n+1$. (14) entails (13) by ω-induction because, for all $n \geqslant 0$,

$$\text{give abstraction of (give } n \text{ then } A) \lesssim_{\text{MAY}} \text{generate then give application of (abstraction of } A\text{) to given data} \,,$$

which can be shown by simulation.

Finally, by combining (12) and (13), the result follows. □

Proposition 6.13 fails for the MAY simulation preorder because the left-hand side evaluates to "abstraction of generate", while the right-hand side evaluates to "abstraction of (give n then A)", for some n : natural, and "generate" is not below "give n then A" in the

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

MAY preorder (neither $\sqsubseteq_{\text{MAY}}$ nor $\lesssim_{\text{MAY}}$) because $i \vdash$ generate $\Downarrow^{\text{MAY}}$ successor of n, while $i \vdash$ give n then $A \Downarrow^{\text{MAY}} n'$ only if $n' = \min(n', n)$.

For the MUST preorder there is the following counterexample.

Proposition 6.14. give (a, b) or give $(b, a) \sqsubseteq_{\text{MUST}}$ give (a, a) ,
where $a =$ abstraction of diverge ; $b =$ abstraction of check true .

The formal proof is involved and requires detailed analysis of action evaluation and termination. We will only sketch the argument. We use experimental restriction, Proposition 5.8. Suppose $i \vdash E$[give (a, b) or give (b, a)] $\downarrow_{\text{MUST}}$. This will imply that we have both $i \vdash E$[give (a, b)] $\downarrow_{\text{MUST}}$ and $i \vdash E$[give (b, a)] $\downarrow_{\text{MUST}}$. Now consider any evaluation $i \vdash E$[give (a, b)] $\Downarrow^{\text{MAY}} t$. a cannot be enacted in the course of the evaluation as this would lead to divergence, so the evaluation must be 'uniform' in a, that is, there is a corresponding evaluation for arbitrary abstractions in place of a. Moreover, the judgement $i \vdash E$[give (a, b)] $\downarrow_{\text{MUST}}$ is also uniform in a. Symmetrically, the second judgement is uniform in a, and we combine these observations to conclude that both judgements are uniform in both a and b, and therefore also hold for a in place of b, $i \vdash E$[give (a, a)] $\downarrow_{\text{MUST}}$, as required.

However, Proposition 6.14 does not hold for the MUST simulation preorder. For all i, we have both $i \vdash$ give (a, b) or give $(b, a) \Downarrow^{\text{MUST}}$ and $i \vdash$ give $(a, a) \Downarrow^{\text{MUST}}$. However, $i \vdash$ give $(a, a) \Downarrow^{\text{MAY}} (a, a) \times \{\}$, and whatever give (a, b) or give (b, a) evaluates to, it contains the abstraction b, which is not below a with respect to the MUST simulation preorder.

Another source of incompleteness is that both simulation preorders are generally too discriminative with regard to data. For example, consider $A \stackrel{\text{def}}{=}$ bind x to v and bind x to v' .

$$A \sqsubseteq_m \text{bind } x \text{ to } v, \qquad A \sqsubseteq_m \text{produce } \{\}, \tag{15}$$

hold because the resulting bindings b=merge($\{x \mapsto v\}$,$\{x \mapsto v'\}$) from A are 'worse' than both the empty bindings, $\{\}$, and the singleton bindings, $\{x \mapsto v\}$. Looking up x in b or in compound bindings built thereof will succeed only if it succeeds with $\{\}$ and $\{x \mapsto v\}$ in place of b. But neither of (15) hold for $\lesssim_m$, because $\widetilde{\lesssim_m}$ relates b to neither $\{\}$ nor $\{x \mapsto v\}$.

The incompleteness for data can be repaired by redefining the $\sim$ operation coinductively: let $\widetilde{R}$ be the largest compatible relation on data (*cf.* Section 5.1) satisfying, whenever $d \,\widetilde{R}\, d'$,

1 d : data implies d' : data, and

2 $d =$ abstraction of A implies $d' =$ abstraction of A' with $A \; R \; A'$.

This definition is sufficiently discriminative because of the repository of data operations from Section 3.6. For instance, the "when _ then _" operation ensures that the truth values are distinguished, by compatibility of $\widetilde{R}$. Note that, if R is compatible on actions, it follows from the Uniformity Requirement 5.1 that its compatible refinement, $\widehat{R}$, on data is included in $\widetilde{R}$. But otherwise the $\widetilde{R}$ relation is difficult to reason about.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

6.4. *Action laws*

Simulation is an excellent tool for establishing equational and inequational algebraic laws about actions, including those from Mosses (1992, Appendix B). Given this collection of action laws, many questions about action equivalences can be resolved by purely (in)equational reasoning. This proof style is advantageous because it enables users of action semantics to reason about actions at a higher level than the details of the operational semantics.

Below we list a selection of (in)equational laws; all follow by straightforward simulation arguments. First some equational laws about the action primitives:

Action law 1. For all actions A,

(1) check true $\simeq$ give () $\simeq$ choose () $\simeq$ produce $\{\}$.

(2) enact abstraction of A $\simeq$ give () then (produce $\{\}$ hence A) .

(3) enact application of (closure abstraction of A) to given data $\simeq$ A .

The recursion combinator is a fixpoint operator:

Action law 2. unfolding A $\simeq$ A @ unfolding A .

The action combinators satisfy equational algebraic laws:

Action law 3.

(1) furthermore, unfolding *are idempotent*.

(2) or *is associative, commutative, and idempotent; its unit is* check false.

(3) and, and then *are associative; their unit is* check true.

(4) then *is associative; its unit is* regive.

(5) hence *is associative; its unit is* rebind.

(6) erratic or *is associative, commutative, and idempotent*.

(The derived combinator "erratic or" is defined in Section 3.5.)

There are also distributivity laws, which we omit.

Moreover, properties of the choice operators can be characterised by inequational laws: "diverge" and "fail" are bottom elements in the MAY preorder and "or" and "erratic or" are the least upper bound operation. "diverge" is also the MUST bottom element and "erratic or" is the greatest lower bound operation for the MUST preorder.

7. Functional program equivalences

In this section we apply the action theory outlined in Section 6.4 to reason about the example functional language from Section 2. Based on our action semantic description, we sketch a soundness proof of an inequational proof system for the language. Secondly, we extend the language with a nondeterministic choice operator and show how to reason about it. Finally, we discuss further applications.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

7.1. *Inequational proof system*

Consider an inequational proof system for our functional language (assume variable substitution, $_[_/_]$, and free and bound variables are defined in the usual way).

(reflexivity)	$M \sqsubseteq M$	
(transitivity)	$\dfrac{M \sqsubseteq P \ , \ \ P \sqsubseteq N}{M \sqsubseteq N}$	
(substitution)	$\dfrac{P \sqsubseteq Q}{M[P/x] \sqsubseteq M[Q/x]}$	
(α)	$\lambda x.M = \lambda y.M[y/x]$	*
(β)	$(\lambda x.M)Q = M[Q/x]$	*
(β_{true})	**if true then** P **else** $Q = P$	
(β_{false})	**if false then** P **else** $Q = Q$	
(η)	$\lambda y.(\lambda x.M)y = \lambda x.M$	†
(fixpoint)	**rec** $x.M = M$[**rec** $x.M/x$]	*
(recursion induction)	$\dfrac{M[P/x] \sqsubseteq P}{\textbf{rec}\ x.M \sqsubseteq P}$	*

With the usual provisons on free and bound variables:

* free variables in the term being substituted for x must not become bound.

† y is not free in $\lambda x.M$.

Let the $\sqsubseteq$ partial order be defined inductively by this proof system. $M = N$ abbreviates $M \sqsubseteq N \ \wedge \ N \sqsubseteq M$. The proof system is sound with respect to the partial order on functional language terms yielded by our action semantic description of the language.

Theorem 7.1. (Soundness) $M \sqsubseteq N$ implies evaluate $M \ \lesssim_{\text{cvx}}$ evaluate N.

The proof proceeds exactly as corresponding soundness proofs based on conventional denotational semantics. We need two lemmas. First, a functional term only depends on the bindings of its free variables.

Lemma 7.2. produce b_1 hence evaluate $M \ \simeq$ produce b_2 hence evaluate M, if b_1, b_2 coincide on the free variables of M, that is, b_1 at $x = b_2$ at x, for all free x in M.

Proof. By structural induction on M. For the case $M =$ **rec** $x.M'$, we apply recursion induction, using Proposition 5.7 (and thereby eschew the conventional continuity argument). □

Second, we have a substitution lemma, which maps syntactic substitution into the semantic concept of bindings.

Lemma 7.3. produce b hence evaluate $M[N/x] \ \simeq$ produce b' hence evaluate M, where $b' \stackrel{\text{def}}{=}$ overlay($\{x \mapsto$ abstraction of (produce b hence evaluate $N)\}$, b) .

Proof. The proof is by structural induction on M, using the previous lemma for the cases $M = \lambda x.M'$, $M =$ **rec** $x.M'$. □

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

Using this lemma, each rule in the proof system maps into a corresponding action rule. It is easy to verify that every rule preserves soundness, hence establishing the soundness theorem.

7.2. *Nondeterminism*

Ong (1993) extends a functional language with a nondeterministic construct "+" (not to be confused with arithmetic addition). We can describe this by means of the (derived) "erratic or" action combinator from Section 3.5.

- Expression = □ | ⟦ Expression "+" Expression ⟧ .

(1) evaluate ⟦ E_1:Expression "+" E_2:Expression ⟧ =
evaluate E_1 erratic or evaluate E_2 .

The functional proof system above remains sound with this extension, because no existing denotations are changed and we used the general nondeterministic action theory to show soundness above. We can also add Ong's laws for +. It is idempotent, commutative and associative, and satisfies some distributive laws. These all follow from corresponding action laws for the "erratic or" combinator. Furthermore, because our equivalence on actions is contextual, we can validate the following law, which Ong shows is not respected by his finer bisimulation equivalence.

$$\lambda x.\Omega \;+\; \lambda x.\lambda y.\Omega = \lambda x.(\Omega + \lambda y.\Omega) \; .$$

We split the proof into MAY and MUST. Each follow by straightforward (in)equational reasoning, using the fact that divergence is bottom and that "erratic or" is the least upper bound for MAY and the greatest lower bound for MUST.

7.3. *Further applications*

We have also studied an extension of our functional language with lazy lists. This has become the prototypical application for co-induction on higher-order programming languages, see Gordon (1995a) and Pitts (1994) for instructive examples. Via the language's action semantic description, these examples can all be rephrased and solved at the level of actions by means of simulation on actions.

Simulation is generally not very informative about recursion, *e.g.* it is of no assistance for proving that the **Y** combinator, $\mathbf{Y} \stackrel{\text{def}}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, is the least fixpoint combinator:

$$\mathbf{rec}\ x.M = \mathbf{Y}\ (\lambda x.M) \; .$$

Instead, such results about recursion can be proved, rather primitively, by the finite restriction characterisation.

Action semantics is useful for reasoning about language implementations. In the Actress compiler generator, intermediate action notation representations of programs are optimised and translated by transformations (Moura and Watt 1994). The verification of these transformations would be an interesting task. However, some transformations

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

introduce imperative action constructs and are beyond the scope of the theory developed here.

There are many other potential applications of our functional/declarative action theory that we have not addressed yet, *e.g.* applications involving countable nondeterminism, dynamic scope, or arbitrary abstract data types. Hitherto we have furnished the theory of actions by attempting to match existing theory about functional programming. The results are promising. Although the action semantic description is not fully abstract, it is able to deliver most results of practical interest as straightforwardly as existing reasoning techniques.

Compared to customised functional theories working directly on the language syntax, translation into actions is, of course, more work. Moreover, the verbose syntax of actions makes reasoning notationally cumbersome in some cases. We believe that these disadvantages are outweighed by the generality of the action semantic approach, since our techniques can be applied to any programming language whose action semantics is expressed in terms of the action notation considered here.

8. Conclusion

8.1. *Summary*

We have presented a semantic theory for a substantial functional/declarative subset of action notation, involving computational features such as unbounded nondeterminism, higher-order functions, dynamic scope and abstract data types. The semantic theory is developed by operational means, based on an evaluation semantics for actions.

This is first of all a contribution to the theory of action semantics and is the first comprehensive study of semantic equivalences for action notation. But our results for (unbounded) nondeterminism and abstract data types should also be applicable to other formalisms with these features.

The main technical result of this paper is that the simulation methods are sound. The challenge has been to define evaluation semantics, simulation and auxiliary notions, such as uniformity of data operations and compatible refinement, so that Howe's method for proving congruence of simulation orderings applies.

8.2. *Future work*

The operational reasoning presented in this paper is based on an evaluation semantics. This style of semantics cannot adequately model interleaved and parallel computation, which is present in full action notation. Therefore, we have also studied a substitution-based extension of the small-step reduction semantics from Lassen (1995) to functional/declarative actions. All the theory presented above can be reworked for this reduction semantics. In order to make the simulation techniques scale up to adequately deal with interleaving computation, we are investigating theories of small-step simulation on reductions.

Here we have studied untyped action semantics, but it would be interesting to combine our work with the 'soft types' provided by the facet notation in Mosses (1992); see also

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

Doh and Schmidt (1994). The facet notation makes it possible to formulate a richer algebra of action laws. Strongly typed action semantics has also been used, especially in compiler generators, *e.g.* Ørbæk (1994). It should be straightforward to construct typed variants of our theory in the style of Gordon (1994).

8.3. *Discussion*

Most other work on denotational semantics operate with metalanguages based on lambda calculus, which has a rich denotational and operational theory. However, they have been successfully applied in relatively few large-scale semantic descriptions of realistic programming languages. Some deficiencies with regard to modularity of denotational semantics are addressed by Moggi's monadic metalanguage, which encapsulates different notions of computation (Moggi 1991). Action semantics is an alternative framework with good pragmatic properties, which derive from the design of action notation. It is a comparatively large notation and is effectively a superset of conventional lambda-based metalanguages. The versatility of action notation makes the task of developing its semantic theory challenging, but we believe that the result will be informative and useful.

Originally, actions were specified algebraically as 'abstract semantic algebras' (Mosses 1983). In Section 6.4 we outlined how the operational theory for actions entails such algebraic action laws. It seems difficult to assess an algebraic semantics without the complementary operational formalisation of dynamic behaviour. Alternatively, one could consider a domain-theoretic approach to the semantics of action notation. However, this will be complicated by the presence of (unbounded) nondeterminism; and a domain-theoretic semantics for full action notation will be very complex. An operational semantics offers a simpler definition of computational behaviour, and we believe that it is also a simpler foundation for the semantic theory.

Our aspiration is to obtain a general action theory for reasoning about a large class of programming languages. The main target for action semantics and for action theory is real-world programming languages in practical use. To this end our work still needs to be extended to encompass a larger part of action semantics, especially the imperative facet. However, the present work is an important step and holds promise for a fully-fledged action theory.

Appendix A. Unified algebra

Unified algebra (Mosses 1989) is the algebraic specification framework normally used together with action semantics. We specify all entities, both syntax and data, in unified algebras. Throughout this paper, equality, $=$, means equality in the (initial) algebraic theory, not syntactic identity. For brevity, our algebraic specifications omit specification of signatures, model constraints and module structure.

Unified algebras treat both individuals and sorts as values, so operations can be applied to sorts as well as to individuals. Moreover, no distinction is made between a singleton sort and its only element. Sorts are partially ordered by subsort inclusion. $S \mid S'$ denotes sort union of sorts S, S'. Individual inclusion, $S : S'$, means both that S is an individual

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

and that S is a subsort of S', written $S \leqslant S'$. We write $S_1,\ldots,S_n : S$ for the conjunction $S_1 : S; \ldots; S_n : S$. (Unified algebras also have a bottom sort nothing and sort intersections, $S \,\&\, S'$. For simplicity of presentation we omit these.)

Axioms of specifications are Horn clauses involving equality, sort inclusion and individual inclusion. All operations are monotone in the subsort ordering. Models of unified algebra specifications are join semi-lattices, equipped with a distinguished subset of individuals, together with monotone functions. Specifications always have initial models.

Consider the specification of sort natural.

- natural = 0 | successor of natural .
- 0 : natural .
- successor of _ :: natural → natural (*total*, *injective*) .

The specification of sort natural in the first clause exploits the unified treatment of sorts and values: the whole sort is a valid argument to "successor of". The clause may be read as a grammar with sort natural as nonterminal, and "0" and "successor of" as terminals.

And we define appropriate operations, for instance "min" that computes the minimum of two natural numbers.

- min(_,_) :: natural, natural → natural (*total*, *associative*, *commutative*, *unit is* 0) .

(1) m, n : natural ⇒ min(successor of m, successor of n) = successor of min(m, n) .

Operations may be specified in prefix, postfix, infix, or general 'mixfix' notation. The placeholder _ marks argument positions in the operation. The functionality natural, natural → natural abbreviates the clause:

(2) min(natural, natural) ⩽ natural .

The attributes *total, associative, etc.* further specify the functionality, but also abbreviate appropriate clauses. For instance, *total* abbreviates the fact that all individuals are mapped to individuals:

(3) m, n : natural ⇒ min(m, n) : natural .

(The other attributes have the expected definitions; their precise definitions can be found in Mosses (1992, Appendix F).)

The operation will implicitly map arguments outside the defined argument domains to vacuous sorts (which we think of as undefined values). Suppose natural ⩽ data. We can overload "min" to become a partial data operation.

- min(_,_) :: data, data → data (*partial*) .

Here *partial* relaxes (3) to allow individuals to be mapped to vacuous sorts:

(4) d, d' : data ; d'' : min(d, d') ⇒ min(d, d') : data .

Acknowledgements

I owe thanks to Peter Mosses for his guidance during this work and to Andy Pitts, Andy Gordon, and Dave Sands for inspiring discussions. Thanks are also due to Luke Ong who made (Ong 1992) available to me. Finally, I am grateful for the referees' comments,

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

which helped me to improve the final version of the paper, especially the presentation of action semantics.

References

Abramsky, S. (1990) The lazy lambda calculus. In: Turner, D. (ed.) *Research topics in functional programming*, Addison-Wesley 65–116.

Buhl, J. (1994) Communicative action semantics. M.Sc. dissertation, Dept. of Computer Science, Univ. of Aarhus.

DeNicola, R. and Hennessy, M. (1984) Testing equivalences for processes. *Theoretical Computer Science* **34** 83–133.

Doh, K.-G. (1993) Action semantics: A tool for developing programming languages. In: *Proceedings of InfoScience'93* (ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/Doh93IS)

Doh, K.-G. and Schmidt, D. (1994) The facets of action semantics: Some principles and applications. In: Mosses, P. D. (ed.) Proceedings of the First International Workshop on Action Semantics, number NS-94-1 in BRICS Notes Series, Dept. of Computer Science, Univ. of Aarhus 1–15.

Gordon, A. D. (1994) *Functional Programming and Input/Output*, Cambridge University Press.

Gordon, A. D. (1995a) Bisimilarity as a theory of functional programming (mini-course). BRICS Notes Series NS-95-3, BRICS, Dept. of Computer Science, Univ. of Aarhus.

Gordon, A. D. (1995b) A tutorial on co-induction and functional programming. In: *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Springer Workshops in Computing 78–95.

Hansen, B. S. and Toft, J. U. (1994) The formal specification of ANDF, an application of action semantics. In: Mosses, P. D. (ed.) Proceedings of the First International Workshop on Action Semantics, number NS-94-1 in BRICS Notes Series, Dept. of Computer Science, Univ. of Aarhus 34–42.

Howe, D. J. (1989) Equality in lazy computation systems. In: *4th LICS*, IEEE.

Lassen, S. B. (1995) Basic Action Theory. Technical Report RS-95-25, BRICS, Dept. of Computer Science, Univ. of Aarhus.

Moggi, E. (1991) Notions of computation and monads. *Information and Computation* **93** 55–92.

Mosses, P. D. (1983) Abstract semantic algebras! In: *Formal Description of Programming Concepts II*, IFIP.

Mosses, P. D. (1989) Unified algebras and institutions. In: *4th LICS*, IEEE 304–312.

Mosses, P. D. (1992) *Action Semantics*, Cambridge University Press.

Mosses, P. D. (1996) Theory and practice of Action Semantics. In: MFCS'96. *Springer-Verlag Lecture Notes in Computer Science* **1113**.

Mosses, P. D. and Watt, D. A. (1993) Pascal action semantics. (ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/MossesWatt93DRAFT/)

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

Ong, C.-H. L. (1992) Concurrent lambda calculus and a general pre-congruence theorem for applicative bisimulation (unpublished preliminary version).

Ong, C.-H. L. (1993) Non-determinism in a functional setting. In: *8th LICS*, IEEE.

Ørbæk, P. (1994) OASIS: An optimizing action-based compiler generator. In: CC'94. *Springer-Verlag Lecture Notes in Computer Science* **786** 1–15.

Pitts, A. M. (1994) Inductive and co-inductive techniques in the semantics of functional programs. Course held at BRICS, Dept. of Computer Science, Univ. of Aarhus.

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press

Reynolds, J. C. (1983) Types, abstraction, and parametric polymorphism. In: *Information Processing '83*, North-Holland 513–523.

Smith, S. (1992) From operational to denotational semantics. In: MFPS'91. *Springer-Verlag Lecture Notes in Computer Science* **598** 54–76.

Ulidowski, I. (1992) Equivalences on observable processes. In: *7th LICS*, IEEE.

Watt, D. A. (1996) Standard ML action semantics, version 0.4 (unpublished interim draft, covering only the ML core language).

https://doi.org/10.1017/S0960129597002417 Published online by Cambridge University Press