

Unique Fixed Point Induction for McCarthy’s *Amb*

Søren B. Lassen¹ and Andrew Moran²

¹ University of Cambridge Computing Laboratory,
Pembroke Street, Cambridge CB2 3QG, United Kingdom
`Soeren.Lassen@cl.cam.ac.uk`

² Department of Computing Science,
Chalmers University of Technology, 412 96 Göteborg, Sweden
`Andrew.Moran@cs.chalmers.se`

Abstract. We develop an operational theory of higher-order functions, recursion, and fair non-determinism for a non-trivial, higher-order, call-by-name functional programming language extended with McCarthy’s *amb*. Implemented via fair parallel evaluation, functional programming with *amb* is very expressive. However, conventional semantic fixed point principles for reasoning about recursion fail in the presence of fairness. Instead, we adapt higher-order operational methods to deal with fair non-determinism. We present two natural semantics, describing may- and must-convergence, and define a notion of contextual equivalence over these two modalities. The presence of *amb* raises special difficulties when reasoning about contextual equivalence. In particular, we report on a challenging open problem with regard to the validity of bisimulation proof methods. We develop two sound and useful reasoning methods which, in combination, enable us to prove a rich collection of laws for contextual equivalence and also provide a unique fixed point induction rule, the first proof rule for reasoning about recursion in the presence of fair non-determinism.

1 Introduction

First introduced in [12], McCarthy’s *amb*, or *ambiguous choice*, has a simple informal operational description: evaluate each operand of the choice in fair parallel and accept the first to terminate as the result. This process will terminate when either operand does, but can only loop when both operands do.

Fair parallel evaluation is a powerful programming idiom and functional programming with *amb* is very expressive. It subsumes many other, more commonly studied non-deterministic and parallel functional language extensions – *e.g.*, erratic choice, countable choice, and parallel or – all straightforwardly encoded in terms of *amb* – but it is also substantially more difficult to model than each of these. In particular, conventional semantic fixed point principles for reasoning about recursion fail in the presence of fairness: *amb* is not even monotonic, let alone continuous, with respect to any domain-theoretic partial order respecting total correctness.

Instead, we adapt higher-order operational methods to give an operational theory of algebraic datatypes, higher-order functions, recursion, and fair non-determinism for a call-by-name functional programming language extended with McCarthy’s *amb*. This is an interesting and challenging application of such techniques, since we cannot look to domain theory for inspiration. The theory contains not only the expected equational laws, but also a proof rule for proving properties of recursive programs containing *amb*: *unique fixed point induction*.

The paper is organised as follows. Section 2 surveys related work. Section 3 presents the language and its operational semantics in the form of natural semantics rules for *may-convergence* and *must-convergence* relations. Section 4 defines contextual equivalence. Our first major result is that contextual equivalence includes a simpler Kleene equivalence relation. It is used to establish many expected equational laws. Section 5 introduces simulation and bisimulation relations. Section 6 is devoted to a cost-sensitive simulation relation, based upon an operational notion of cost. Our second major result is that this is a congruence and, hence, included in contextual equivalence. We develop a tick algebra for the language and establish the powerful unique fixed point induction proof rule. Unique fixed point induction allows us to prove two terms contextually equivalent by exhibiting a *single* program context. We demonstrate its use in section 7, where it is used to prove properties of bottom-avoiding merge.

2 Related Work

Traditionally, one gives meaning to non-deterministic constructs via *powerdomain* constructions, domain-theoretic analogues of the powerset operator; see [21] for an accessible survey. The denotational approach encounters well-known problems when attempting to model McCarthy’s *amb*. For example, the Egli-Milner ordering is a natural preorder to consider (it combines domain-theoretic analogues of may- and must-behaviours), but McCarthy’s *amb* is not even monotonic with respect to this ordering, so the theory breaks down. The only attempt at a denotational semantics for McCarthy’s *amb*, due to Broy [3], is developed for a first-order language only, and it is rather cumbersome and complex. Possibly, Moschovakis’ powerstructures [16] are an alternative candidate for a model of a higher-order language with fair non-determinism.

Higher-order co-inductive operational techniques, based upon Howe’s method for proving congruence of bisimulation equivalences for functional languages [7], have been applied to non-deterministic functional languages: Howe [7] and Ong [17] consider erratic choice (where one operand is chosen and then evaluated, while the other is discarded), showing that notions of simulation pre-orders and bisimulation equivalences are congruences. Lassen and Pitcher [11] prove congruence of a space of simulation pre-orders and bisimulation equivalences for countable choice (which returns a random natural number). Quite different operational techniques are used by Moran, Sands, and Carlsson [15] to develop an equational theory for call-by-need and erratic choice. Both erratic and countable choice are *sequential* forms of non-determinism; it is not immediate how to ex-

tend any of these methods to the fair non-determinism exhibited by McCarthy's *amb*.

Here we extend these methods by using an alternative to Howe's relational construction (from [9]), and by introducing costs (following Sands' improvement theory for deterministic languages [18]). The unique fixed point induction rule is adapted from [18]. Our adaption inspired a similar rule in [15], where it is used in an extensive treatment of a theory for erratic choice (that respects sharing) for the Fudgets' stream processor calculus.

This paper reports on results from the authors' dissertations [10, 13]. Mostly, proofs are sketched or omitted; detailed proofs may be found in [13].

3 The Operational Semantics

We begin by introducing the operational semantics for a call-by-name λ -calculus plus McCarthy's *amb*. We give natural semantics for both *may-convergence* and *must-convergence*. We need to describe must-convergence behaviour since *amb* is distinguished from erratic choice by its must-convergence (or equivalently, divergent) behaviour. In anticipation of the development of a cost-sensitive operational theory, we attribute costs to the two natural semantics.

The terms in the language are of the form:

$$\begin{aligned} x, y, z \in Variable \quad K \in Constructor \quad op \in Primitive \\ M, N ::= x \mid \lambda x.M \mid M N \mid M [] N \mid \ulcorner n \urcorner \mid M op N \mid K M_1 \cdots M_n \\ \mid \text{fix } M \mid \text{case } M \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n \end{aligned}$$

Constructor is a set of names, disjoint from *Variable*, which may not be bound or alpha converted; we will represent lists by assuming a nullary constructor *nil*, denoted by $[]$, and a binary constructor *cons*, denoted by the infix symbol $(:)$. *Primitive* is the set of standard integer primitives (addition, division, equality, *etc.*); equality and related primitives map pairs of integers to nullary constructors *true* and *false*. For each integer n , we have a distinguished value, written $\ulcorner n \urcorner$. Other values are lambda expressions and constructed values; all values are ranged over by U and V . In addition we have explicit recursion and case expressions. We denote McCarthy's *amb* by the infix symbol $[]$.

For illustration let us see how other non-deterministic operators can be encoded using *amb*. Erratic choice, written as an infix \oplus , makes an initial choice between its two operands. The selected operand is evaluated and the other is discarded. Using *amb*, we can simulate erratic choice between terms M and N thus:

$$M \oplus N \stackrel{\text{def}}{=} ((K M) [] (K N)) \mid$$

where $K \stackrel{\text{def}}{=} \lambda x y.x$, and $\mid \stackrel{\text{def}}{=} \lambda x.x$. Countable choice, also known as random assignment and denoted by $?$, can evaluate to any natural number but cannot diverge. It can be expressed as:

$$? \stackrel{\text{def}}{=} \text{fix } \lambda x. 0 [] (x + 1).$$

$$\begin{array}{c}
\lambda x.M \Downarrow^0 \lambda x.M \quad (Lam_{\Downarrow^n}) \qquad \quad \quad \quad \Uparrow^n \Downarrow^0 \Uparrow^n \quad (Int_{\Downarrow^n}) \\
K M_1 \cdots M_n \Downarrow^0 K M_1 \cdots M_n \quad (Constr_{\Downarrow^n}) \qquad \quad \quad \quad \frac{M \Downarrow^m \Uparrow_i \Uparrow \quad N \Downarrow^n \Uparrow_j \Uparrow}{M \text{ op } N \Downarrow^{m+n} \Uparrow_i \text{ op } \Uparrow_j \Uparrow} \quad (Op_{\Downarrow^n}) \\
\frac{M \Downarrow^n V}{M \Downarrow N \Downarrow^n V} \quad (Amb_{\Downarrow_1^n}) \qquad \quad \quad \quad \frac{N \Downarrow^n V}{M \Downarrow N \Downarrow^n V} \quad (Amb_{\Downarrow_2^n}) \\
\frac{M (\text{fix } M) \Downarrow^n V}{\text{fix } M \Downarrow^n V} \quad (Fix_{\Downarrow^n}) \qquad \quad \quad \quad \frac{M \Downarrow^m \lambda x.M' \quad M'[\Uparrow/x] \Downarrow^n V}{M N \Downarrow^{m+n+1} V} \quad (App_{\Downarrow^n}) \\
\frac{M \Downarrow^m K_j M_1 \cdots M_{m_j} \quad N_j [M_k/x_k]_{k=1}^{m_j} \Downarrow^n V}{\text{case } M \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n \Downarrow^{m+n+1} V} \quad (Case_{\Downarrow^n})
\end{array}$$

Fig. 1. Natural semantics for may-convergence, instrumented with costs.

A more involved example is the “bottom-avoiding” merge operator defined thus:

$$merge \stackrel{\text{def}}{=} \text{fix } \lambda m \text{ xs } ys. \left(\begin{array}{l} \text{case xs of} \\ [] \rightarrow ys \\ z : zs \rightarrow z : m \text{ xs } ys \end{array} \right) \Downarrow \left(\begin{array}{l} \text{case ys of} \\ [] \rightarrow xs \\ z : zs \rightarrow z : m \text{ xs } zs \end{array} \right)$$

It non-deterministically merges two possibly infinite lists. If one of the two lists is finite, every element of the other appears in the output.

The natural semantics rules for may-convergence behaviour is given in figure 1. We write $M \Downarrow^n V$ to mean that closed term M may converge to value V with cost n . The rules in figure 1 describe a leftmost-outermost evaluation strategy to weak head normal form and are straightforward. The constructor and case rules specify that constructors are lazy in our language.

The cost measures the number of function application steps and constructor elimination steps in the computation. These steps involve general substitutions. The cost measure is motivated by technical requirements in the development of the cost-sensitive theory later on. Note that we can recover the usual definition of may-convergence easily: $M \Downarrow V \stackrel{\text{def}}{=} \exists n.M \Downarrow^n V$. We also write $M \Downarrow$ to mean that there exists some value V such that $M \Downarrow V$.

The natural semantics rules for must-convergence behaviour is given in figure 2. We write $M \Downarrow^\alpha$ to mean that closed term M must converge at cost α . The cost is an ordinal which is, roughly, the “supremum” of the costs of converging computations that witness that M must converge (they form a recursive, countably branching tree and α is measured as the height of this tree, hence α is a recursive ordinal; cf. [1, 11]). We can recover simple must-convergence thus: $M \Downarrow \stackrel{\text{def}}{=} \exists \alpha.M \Downarrow^\alpha$.

The two $(Amb_{\Downarrow_i^\alpha})$ rules distinguish *amb* from erratic choice, and this highlights the need for a description of must-convergence behaviour. For an erratic choice to always converge, *both* branches must always converge. For McCarthy’s *amb* to always converge, it is sufficient that at least one operand does so. In [13]

$$\begin{array}{c}
\lambda x.M \downarrow^0 \text{ (Lam}_{\downarrow^0}) \quad \ulcorner n \urcorner \downarrow^0 \text{ (Int}_{\downarrow^0}) \quad K M_1 \cdots M_n \downarrow^0 \text{ (Constr}_{\downarrow^0}) \\
\frac{M \downarrow^\alpha}{M \parallel N \downarrow^\alpha} \text{ (Amb}_{\downarrow^1}) \quad \frac{N \downarrow^\alpha}{M \parallel N \downarrow^\alpha} \text{ (Amb}_{\downarrow^2}) \quad \frac{M \downarrow^{\alpha_1} \quad N \downarrow^{\alpha_2}}{M \text{ op } N \downarrow^{\alpha_1 + \alpha_2}} \text{ (Op}_{\downarrow^\alpha}) \\
\frac{M \text{ (fix } M) \downarrow^\alpha}{\text{fix } M \downarrow^\alpha} \text{ (Fix}_{\downarrow^\alpha}) \quad \frac{M \downarrow^\alpha \quad \forall V. M \downarrow V \implies V \equiv \lambda x. M' \wedge M' [N/x] \downarrow^{\alpha_V}}{M N \downarrow^{(\cup_V \alpha_V + 1) + \alpha}} \text{ (App}_{\downarrow^\alpha}) \\
\frac{M \downarrow^\alpha \quad \forall V. M \downarrow V \implies V \equiv K_j M_1 \cdots M_{m_j} \wedge N_j [M_k/x_k]_{k=1}^{m_j} \downarrow^{\alpha_V}}{\text{case } M \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n \downarrow^{(\cup_V \alpha_V + 1) + \alpha}} \text{ (Case}_{\downarrow^\alpha})
\end{array}$$

Fig. 2. Natural semantics for must-convergence, instrumented with costs.

the \downarrow predicate is shown to be an accurate description of the must convergence behaviour of the fair parallel evaluation semantics of *amb* alluded to in section 1.

When deciding whether or not $M N \downarrow^\alpha$, not only must M converge, but it may only converge to lambda expressions. A similar restriction affects the rule for case expressions. This is due to the fact that we are working with an untyped language. The rules $(App_{\downarrow^\alpha})$ and $(Case_{\downarrow^\alpha})$ are infinitary, since in each case M may converge to countably many different values.

4 The Equational Theory

In this section, we first define a Morris-style contextual equivalence for the language, faithful to both may-convergence and must-convergence behaviour. We then establish a collection of equational laws for contextual equivalence by means of a simpler, more tractable equivalence relation, called Kleene equivalence.

In the following definition, \mathbb{C} ranges over *program contexts* relative to M and N , that is, terms \mathbb{C} with an occurrence of a *hole* $[\cdot]$ such that the terms $\mathbb{C}[M]$ and $\mathbb{C}[N]$, obtained by filling in M and N for $[\cdot]$, are closed.

Definition 1. We define contextual equivalence, \cong , by

$$M \cong N \stackrel{\text{def}}{=} \forall \mathbb{C}. (\mathbb{C}[M] \downarrow \iff \mathbb{C}[N] \downarrow \wedge \mathbb{C}[M] \downarrow \iff \mathbb{C}[N] \downarrow).$$

If we read the may-converge predicate as a simple partial correctness assertion and the must-converge predicate as a simple total correctness assertion, the two conjuncts in the definition assert that contextual equivalence respects both partial and total correctness (*i.e.*, closed contextually equivalent terms satisfy the same correctness assertions). The quantification over contexts makes contextual equivalence the greatest such relation which is closed under contexts: $M \cong N \implies \mathbb{C}[M] \cong \mathbb{C}[N]$.

Contextual definitions are useful for showing two terms to be distinct (all that is needed is a *witnessing* context that exhibits different behaviour for each term), but cumbersome to use for proving equivalence, since the definitions involve quantification over *all* contexts. For this reason, we seek more tractable

characterisations of contextual equivalence, or at least, sound approximations that yield valid methods for establishing contextual equivalences between terms. We shall pursue this quest in greater depth in the next sections, but first let us introduce a simple-minded approximation to contextual equivalence, called *Kleene equivalence*. It relates terms with identical convergence behaviour:

Definition 2. We define Kleene equivalence, \approx , between closed terms by

$$M \approx N \stackrel{\text{def}}{=} (\forall V. M \Downarrow V \iff N \Downarrow V) \wedge M \downarrow \iff N \downarrow.$$

For open terms, $M \approx N$ if and only if $M\sigma \approx N\sigma$ for all closing substitutions σ .

It is often simple to verify that two terms are Kleene equivalent, and surprisingly many fundamental equational laws, e.g. β -laws, are instances of Kleene equivalence.

Our first major result is that Kleene equivalence is included in contextual equivalence:

Lemma 1. $\approx \subseteq \cong$.

The proof is postponed to the next section.

We note that $\approx \neq \cong$. In particular, \approx is not closed under contexts, e.g. $M \approx N$ implies $\lambda x.M \approx \lambda x.N$ only if M and N are identical, whereas $\lambda x.M \cong \lambda x.N$ holds whenever $M \cong N$; and similarly for constructed values.

Some selected equational laws for call-by-name and McCarthy's *amb* are:

$$\begin{aligned} (\lambda x.M) N &\cong M[N/x] && (\beta) \\ \text{case } K_j M_1 \cdots M_{m_j} \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n &\cong N_j [M_k/x_k]_{k=1}^{m_j} && (\text{case-}\beta) \\ \text{fix } M &\cong M (\text{fix } M) && (\text{fix-}\beta) \\ \text{case } \Omega \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n &\cong \Omega && (\text{case-strict}) \\ M \square M &\cong M && (\square\text{-idem}) \\ M \square N &\cong N \square M && (\square\text{-comm}) \\ (L \square M) \square N &\cong L \square (M \square N) && (\square\text{-assoc}) \\ M \square \Omega &\cong M && (\square\text{-}\Omega) \end{aligned}$$

These are all justified by lemma 1. As an example, we prove the validity of (β) . It suffices to prove it for closed terms $\lambda x.M$ and N ; it then follows easily for open terms as well. By (App_{\Downarrow^n}) , whenever $(\lambda x.M) N$ may converge to some V , $M[N/x]$ may also converge to the same V (and *vice versa*). By $(App_{\downarrow\alpha})$, whenever $(\lambda x.M) N$ must converge, so must $M[N/x]$ (and *vice versa*). Therefore $(\lambda x.M) N \approx M[N/x]$, and, by lemma 1, $(\lambda x.M) N \cong M[N/x]$. The other laws of the equational theory follow similarly.

A few other properties of \cong can be derived from its definition: it is an equivalence relation (reflexive, transitive and symmetric), and it is closed under contexts. These properties combined with (β) entail that \cong is substitutive: $M \cong N \wedge M' \cong N' \implies M[M'/x] \cong N[N'/x]$.

5 Simulation

In the previous section we saw several good uses of Kleene equivalence for reasoning about contextual equivalence. But Kleene equivalence is too discriminating when considering programs which may-converge to higher-order values, *i.e.* lambda expressions or (lazy) constructed values. For instance, it cannot be used to establish any interesting algebraic laws for *merge*. The problem is that equivalent terms are required to may-converge to the *same* set of values. If we relax this and instead require that equivalent terms may-converge to suitably related sets of values, we arrive at forms of (bi)simulation equivalence. These are the subject of the remainder of the paper.

In this section we introduce notions of simulation and bisimulation. We state open problems regarding the relationship between (bi)simulation equivalences and contextual equivalence, and we prove that contextual equivalence includes Kleene equivalence.

First, we define a useful relational operator called compatible refinement [4]. Given a term relation R , its compatible refinement, \widehat{R} , relates two terms M and N if they have the same outer syntactic constructor, the components of which are pair-wise related by R . For example, the definition of compatible refinement between values is: (1) $\ulcorner n \urcorner \widehat{R} \ulcorner n \urcorner$, for all n ; (2) $\lambda x.M \widehat{R} \lambda x.N$, if $M R N$; (3) $K M_1 \cdots M_n \widehat{R} K N_1 \cdots N_n$, if $M_i R N_i$ for $1 \leq i \leq n$. A relation R is *compatible* if $\widehat{R} \subseteq R$. Every compatible relation is closed under contexts. A *congruence* is a compatible equivalence relation.

The first use of compatible refinement is in the following definition.

Definition 3. *A relation S between closed terms is a simulation when*

$$M S N \implies \forall \text{closing } \sigma. \forall V. N\sigma \Downarrow V \implies \exists U. M\sigma \Downarrow U \wedge U \widehat{S} V \\ \wedge M\sigma \Downarrow \implies N\sigma \Downarrow.$$

For example, Kleene equivalence, \approx , is a simulation.

The greatest simulation (the union of all simulations) is a pre-order (reflexive and transitive). Let *mutual similarity* be the induced equivalence relation. The question of the congruence of mutual similarity (or of *bisimilarity*, the greatest symmetric simulation) is an open one, and has been studied extensively by Corin Pitcher and the authors. If this question is answered affirmatively, mutual similarity will be included in contextual equivalence. In particular, it will entail that \approx is included in contextual equivalence, lemma 1, because \approx is a symmetric simulation and, hence, is included in mutual similarity.

As mentioned in section 3, *amb* is distinguished from erratic choice by its must-convergence behaviour, imposing quite different obligations on implementations. This may explain why the seemingly small difference in the natural semantics rules defining must-convergence makes *amb* significantly more difficult to reason about. For erratic choice, and even countable choice, a collection of (bi)simulation equivalences that respect both may-convergence and must-convergence behaviour have been proven congruent by variations of Howe's method [7, 17, 11]. These proofs fail for *amb*, for the following technical reason.

In the course of the congruence proofs one shows that a suitable “congruence candidate relation” preserves must-convergence, *i.e.*, for all related terms M and N , $M \Downarrow \implies N \Downarrow$, which is established by rule induction on $M \Downarrow$. For the last three must-convergence rules in figure 2 we find that we need to know about the set of outcomes of may-convergence of the left-most subterm of M . So the induction hypothesis must be suitably strengthened to carry information about may-convergence. This works well for the natural semantics of erratic choice and countable choice, but not for *amb*. In the $(Amb_{\downarrow\uparrow})$ rules the induction hypothesis only tells us about one branch, M_1 say, of a choice expression $M_1 \square M_2$, sufficient for deducing the desired properties with regard to must-convergence of $M_1 \square M_2$ which follows from one branch only, but insufficient for the set of outcomes of may-convergence of $M_1 \square M_2$ which depends on both branches.

One possible solution is first (1) to show an appropriate relationship between the may-convergence behaviour of related terms by rule induction on may-convergence judgements, and then (2) use this to prove the desired relationship between must-convergence behaviours by rule induction on must-convergence judgements. But it turns out that the inherent asymmetry in Howe’s construction of the congruence candidate relation prevents us from linking up (1) and (2). However, the simple nature of Kleene equivalence can be exploited to carry out this programme in the proof of lemma 1, $\approx \subseteq \cong$, if we replace the congruence candidate by a different, symmetric construction, denoted by \approx^{SC} . It is defined inductively thus:

$$\frac{M \approx M' \quad \vec{N} \approx^{\text{SC}} \vec{N}'}{M[\vec{N}/\vec{x}] \approx^{\text{SC}} M'[\vec{N}'/\vec{x}]} \qquad \frac{M \widehat{\approx}^{\text{SC}} N}{M \approx^{\text{SC}} N}$$

where $\vec{N} \approx^{\text{SC}} \vec{N}'$ has the obvious meaning. \approx^{SC} is the smallest compatible and substitutive relation containing \approx . We can prove that \approx^{SC} is a simulation by rule induction on may-convergence and must-convergence judgements; the symmetry of \approx^{SC} is crucial in linking up the may- and must-parts of the proof. It follows (by symmetry) that \approx^{SC} only relates terms with identical must-convergence behaviour. As \approx^{SC} is also closed under contexts, it is included in \cong . By transitivity, we conclude that $\approx \subseteq \cong$.

6 Cost Equivalence

We will now present a cost-sensitive form of simulation, following Sands’ improvement theory for deterministic languages [18] (related to the notion of “expansion” in concurrency, see *e.g.* [2, 20]). We are able to show that the induced simulation equivalence, known as *cost equivalence*, is a congruence. This leads to the introduction of a *tick algebra* as in [18], and finally a proof rule for recursive programs known as *unique fixed point induction*.

To define a cost-sensitive simulation, we use the cost attributes of \Downarrow and \downarrow . This change will enable us to show the congruence of the induced mutual similarity and to establish useful proof rules for it.

Definition 4. A relation S between closed terms is a cost simulation when

$$M S N \implies \forall \text{ closing } \sigma. \forall n, V. N\sigma \Downarrow^n V \implies \exists U. M\sigma \Downarrow^{\leq n} U \wedge U \widehat{S} V \\ \wedge \forall \alpha. M\sigma \Downarrow^\alpha \implies N\sigma \Downarrow^{\leq \alpha}$$

where $\Downarrow^{\leq n}$ and $\Downarrow^{\leq \alpha}$ have the obvious meanings.

We denote the resulting mutual cost similarity equivalence relation, or *cost equivalence*, by \doteq .

In the remainder of this section we develop the equational theory of \doteq and we establish bisimulation proof rules for reasoning about \doteq . First, we establish some basic equational laws for use in equational reasoning in later examples. Similarly to the equational theory of \cong in section 4, most of the laws for \doteq equate Kleene equivalent terms but now only those with identical costs. Such terms are easily seen to be cost equivalent (the “cost Kleene equivalence” relation between such terms is a symmetric cost simulation and, hence, is included in \doteq).

With the exception of (β) and $(\text{case-}\beta)$ the laws presented in section 4 are valid for \doteq in place of \cong . Where it does not lead to confusion, we will refer to the \doteq variants by the same name as their \cong counterparts.

It is often the case that two terms are “almost” cost equivalent; that is, they always differ in may- and must-cost by the same fixed amount. For example, $(\lambda x.M) N$ and $M[N/x]$ *always* differ by exactly one unit of cost. If we had some syntactic way of slowing the right-hand side down, we could write the relationship between the two as a cost equivalence. To this end, we introduce the “tick”, written \surd , which we will use to add a dummy step to a computation. Now we can write:

$$(\lambda x.M) N \doteq \surd M[N/x] \quad (\surd\text{-}\beta)$$

There is a similar law for case expressions. We can define \surd within the language by superfluous application: $\surd M \stackrel{\text{def}}{=} \lambda x.M$. Clearly, \surd adds one unit to the cost of evaluating M without otherwise changing its behaviour. Note that by (β) :

$$\surd M \cong M \quad (\text{erase-}\surd)$$

There are a number of useful and easily verified laws for rearranging ticks, including:

$$\begin{aligned} \surd(M \parallel N) &\doteq (\surd M) \parallel (\surd N) && (\surd\text{-}\parallel\text{-dist}) \\ (\surd M) N &\doteq \surd(M N) && (\surd\text{-float-apply}) \\ \text{case } \surd M \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n & && \\ &\doteq \surd \text{case } M \text{ of } \{K_i x_1 \cdots x_{m_i} \rightarrow N_i\}_{i=1}^n && (\surd\text{-float-case}) \end{aligned}$$

The equational laws are collectively known as the tick algebra.

The definitions of the cost measures and of cost equivalence are carefully designed to enable us to prove that cost equivalence is a congruence.

Theorem 1. \doteq is a congruence.

This in turn entails that \doteq is included in \cong . The congruence proof is inspired by the proof of congruence of a cost equivalence relation for a deterministic call-by-value language in [9]. Again, the proof follows the outline of Howe’s method but, here, the congruence candidate relation is, basically, the smallest compatible and *transitive* relation containing \doteq . This is a symmetric construction, a fact that plays a crucial role in the proof, as it did for lemma 1.

The original congruence proof in [9] was designed to facilitate the derivation of Sangiorgi’s bisimulation up to context proof rule [19] and, eventually, Sands’ improvement theorem [18] for cost equivalence. We now present analogous results which can be derived from the congruence proof for our language.

The following proof rule is very useful indeed; it allows us to prove equivalences between recursive programs. Sands has a similar rule in his call-by-name improvement theory [18], which he calls improvement induction, and [14] presents an improvement induction principle for deterministic call-by-need. We dub this proof rule *unique fixed point induction*.

Theorem 2 (Unique Fixed Point Induction). *For all term contexts \mathbb{C} , the following is a valid proof rule:*

$$\frac{M \doteq \check{\mathbb{C}}[M] \quad N \doteq \check{\mathbb{C}}[N]}{M \doteq N}$$

With the help of unique fixed point induction, it is no longer necessary to prove something about M ’s and N ’s behaviour in *all* contexts to prove them contextually equivalent. In most cases, it is enough to exhibit a *single* context and perform some simple equational reasoning.

We have used unique fixed point induction in combination with the (*erase- \checkmark*) law and the inclusion of cost equivalence in contextual equivalence to prove many standard examples of contextual equivalence from deterministic languages, for example the dinaturality law for fixed points and the many monad laws for streams from *e.g.* [22, 5].

7 Bottom-Avoiding Merge

We now prove that the implementation of *merge* is bottom-avoiding:

$$\text{merge } xs \ \Omega \cong xs \ \# \ \Omega.$$

Here, $\#$ is standard list concatenation, or append, and Ω is the always divergent term. Let \mathbb{C} be the (variable-capturing!) context `case xs of {[] → Ω | z : xs → z : []}`. Unfolding *merge xs Ω* and simplifying with (*case-strict*) and (\square - Ω) yields

$$\text{merge } xs \ \Omega \doteq {}^3\check{\mathbb{C}}[\text{merge } xs \ \Omega]$$

and unfolding $xs \ \# \ \Omega$ gives us $xs \ \# \ \Omega \doteq {}^o {}^3\check{\mathbb{C}}[xs \ \# \ \Omega]$. The desired result follows by unique fixed point induction and the inclusion $\doteq \subseteq \cong$.

Merge is also commutative, associative and has the empty list as left and right unit. See [13] for proofs of these properties and other examples. Similar lemmas may be found in [8], but there one needs to “take limits” to include streams. Streams are included here without any extra effort on our part.

References

1. K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.
2. S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. In *TACS '91*, LNCS 526. Springer, 1991.
3. M. Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45(1):1–61, 1986.
4. A. D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge U. P., 1994.
5. A. D. Gordon. Bisimilarity as a theory of functional programming. In *MFPS XI*, ENTCS 1. Elsevier, 1995.
6. A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge U. P., 1998.
7. D. Howe. Proving congruence of bisimulation in functional programming. *Information and Computation*, 124(2):103–112, 1996.
8. J. Hughes and J. T. O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *Functional Programming, Glasgow, 1989*, Workshops in Computing. Springer, 1989.
9. S. B. Lassen. Relational reasoning about contexts. In Gordon and Pitts [6], pages 91–135.
10. S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Dept. of Computer Science, University of Aarhus, May 1998.
11. S. B. Lassen and C. S. Pitcher. Similarity and bisimilarity for countable non-determinism and higher-order functions. In *HOOTS II*, ENTCS 10. Elsevier, 1998.
12. J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.
13. A. K. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Sept. 1998.
14. A. K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL '99*. ACM, 1999.
15. A. K. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, LNCS 1594. Springer, 1999.
16. Y. N. Moschovakis. A game-theoretic, concurrent and fair model of the typed lambda-calculus, with full recursion. In *CSL '97*, LNCS 1414. Springer, 1998.
17. C.-H. L. Ong. Non-determinism in a functional setting. In *LICS '93*. IEEE, 1993.
18. D. Sands. Improvement theory and its applications. In Gordon and Pitts [6], pages 275–306.
19. D. Sangiorgi. On the bisimulation proof method. In *MFCS '95*, LNCS 969, pages 479–488. Springer, 1995.
20. D. Sangiorgi and R. Milner. The problem of “weak bisimulation up to”. In *CONCUR '92*, LNCS 789. Springer, 1992.
21. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.
22. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.