Evaluation Logic[*]

Andrew M. Pitts[†]
University of Cambridge Computer Laboratory
Cambridge CB2 3QG England

Abstract

A new typed, higher-order logic is described which appears particularly well fitted to reasoning about forms of computation whose operational behaviour can be specified using the *Natural Semantics* style of structural operational semantics [5]. The logic's underlying type system is Moggi's *computational metalanguage* [11], which enforces a distinction between computations and values via the categorical structure of a strong monad. This is extended to a (constructive) predicate logic with modal formulas about evaluation of computations to values, called *evaluation modalities.* The categorical structure corresponding to this kind of logic is explained and a couple of examples of categorical models given.

As a first example of the naturalness and applicability of this new logic to program semantics, we investigate the translation of a (tiny) fragment of Standard ML into a theory over the logic, which is proved computationally adequate for ML's Natural Semantics [10]. Whilst it is tiny, the ML fragment does however contain both higher-order functional and imperative features, about which the logic allows us to reason without having to mention global states explicitly.

[*] This article appears in: G. Birtwistle (ed.), *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing (Springer-Verlag, Berlin, 1991), pp 162–189.

[†] Research supported by the CLICS project (ESPRIT BR Action nr 3003).

1 Introduction

Higher-order metalogics based on typed lambda calculi (such as Scott's LCF [20] and Plotkin's formalizations of domain-theoretic denotational semantics [19]) have been used to give semantics to programming languages via formal translations of programming language syntax into the types and terms of the metalogic. The basic features of such translations are their compositionality (i.e. the translation of a compound program expression depends only on the translations of its subexpressions) and that they adequately capture (via provability within the metalogic) the intended operational behaviour of program expressions. Armed with such a translation, amongst other things we can—at least in theory—use the metalogic to reason formally about program behaviours. The ease with which this can be done in practice depends partly on the 'naturalness' of the translation, which in turn depends on how well-fitted the logical forms of expression permitted by the metalogic are to programming language features and their operational semantics.

In this paper we will describe the core of a new metalogic, called *Evaluation Logic*, which appears particularly well fitted to reasoning about forms of computation that can be specified using a style of operational semantics known as *Natural Semantics*. The latter defines the behaviour of the phrases of a programming language via relations such as

$$\mathit{State}, \mathit{Phrase} \Rightarrow \mathit{State}', \mathit{Value}$$

which are inductively defined by rules reflecting the *structure* of program phrases. This style of operational semantics is a particular case of the structural approach of Plotkin [18]. It was developed independently in the context of intuitionistic type theory by P. Martin-Löf (see [15]), and has been further refined and developed by Milner, Kahn [5] and others. A large-scale example of Natural Semantics is provided by the official definition of the Standard ML language [10].

The starting point of the ideas described in this paper is the recent work by Moggi [11, 12, 13] making use of the categorical notion of a *strong monad* as a powerful organizing tool in the denotational semantics of programming languages. (See Gunter and Scott [4] and Mosses [14] for a survey of existing techniques in this area.) Roughly speaking, Moggi's viewpoint is that particular notions of computation can be modelled by various monads T on suitable categories of semantic domains: if datavalues of a particular type are modelled by a domain D, then the denotations of computations of data of that type lie in the domain $T(D)$. The efficacy of this viewpoint is borne out in Moggi's work not only by many concrete examples of monads, but also by the fact that these examples can be built up in a modular way by applying monad constructors corresponding to different features of computations. Instead of building new monads from old using monad constructors, one might consider *axiomatizing* extra, computation-related properties of a single strong monad within the framework of a suitable logic. It is the question of what is a suitable logic for doing this which is addressed in this paper.

There already exists an elegant equational logic of typed terms corresponding to the notion of strong monad (in the same way that the simply-typed lambda calculus corresponds to cartesian closedness). This is Moggi's *computational lambda calculus*, which we review in Section 2. As well as product and function types, this calculus contains *computation types*, $T\sigma$, with two associated term-forming operations capturing the structure of a strong monad. The first operation associates to a term M of type σ a term $[M]$ of type $T\sigma$, whose intended meaning is 'the computation which immediately evaluates to the value M'. The second operation associates to terms $E{:}T\sigma$ and $F(x){:}T\sigma'$ (the second depending upon a variable $x{:}\sigma$), a term

$$\mathit{let}\, x{\Leftarrow}E\ \mathit{in}\, F(x)$$

of type $T\sigma'$, intended to denote a basic form of sequential composition: '*first* evaluate E, bind the result to the parameter x and *then* evaluate $F(x)$'.

Such informal statements about evaluation of computations—giving the intended interpretation of computation terms—are not captured directly in Moggi's computational lambda calculus, which instead gives certain basic *equations* between computations analogous to beta and eta conversion for lambda terms. In this paper we will extend the computational lambda calculus to a constructive predicate logic which permits the formulation of statements about *evaluation of computations to values*, and which accordingly we call *Evaluation Logic.* This is achieved by means of *evaluation modalities* which to each formula $\phi(x)$ containing a free variable $x{:}\sigma$, and to each term $E{:}T\sigma$, assign formulas

$$[x{\Leftarrow}E]\phi(x) \qquad \text{and} \qquad \langle x{\Leftarrow}E\rangle\phi(x)$$

in which x becomes a bound variable. The intended meaning of the first formula is 'if E evaluates to x, then necessarily $\phi(x)$ holds', whilst the intended meaning of the second is 'it is possible for E to evaluate to an x for which $\phi(x)$ holds'. These evaluation modalities have as derived forms predicates asserting evaluation of computations to values, and convergence and divergence of computations; they can also be used to formulate partial and total correctness statements in a natural way—see Remark 3.2.8.

The evaluation modalities and their rules of inference are described in Section 3. Their presence makes Evaluation Logic reminiscent of Dynamic Logic (see Kozen and Tiuryn [7] for a survey of the latter). Indeed the forms of modalities which appear in Dynamic Logic are the particular cases of evaluation modalities with σ the unit type 1 (the type containing a unique element up to equality). However, the motivation for formulating Evaluation Logic came more from the Natural Semantics style of operational semantics mentioned above. In Section 4 we give a simple example of translating programming language features into an

Evaluation Logic *theory* adequately capturing operational behaviour specified in Natural Semantics. The example concerns a fragment of Standard ML containing both functional features (higher-order recursive function declarations) and imperative features (assignable global variables).

Metatheoretical conventions

Evaluation Logic, and the computational lambda calculus over which it is based, contain several unfamiliar variable-binding operations. We deal with these in a uniform way by adopting the increasingly common device (advocated by Aczel, Klop [6] and Martin-Löf [15] amongst others) of using a *higher-order* metalanguage to specify the syntax of object-language expressions. For our purposes here, it is sufficient to use a typed λ-calculus over a single ground type EXP (the type of object expressions), with meta-terms identified up to $\alpha\beta\eta$-conversion. Lambda abstraction in this metalanguage will be denoted by $(x)e$. Meta-application will be denoted by $e(e')$, with a multiple application such as $e(e')(e'')$ abbreviated to $e(e', e'')$. The result of substitution of a meta-term e for a meta-variable x throughout a meta-term e' will be denoted $e'(e/x)$.

In this way, the only variable binding takes place via lambda abstraction in the meta-language, and whilst object-language expressions may contain object-variables, no concept of a *free* object-variable is needed.

Acknowledgements

The work described here has benefited greatly from discussions with E. Moggi, upon whose work it builds, and also from discussions with the other members of the CLICS Project in Cambridge.

2 Computational Lambda Calculus

In this section we review, in a semi-formal style, Moggi's computational metalanguage [11, 13], which adds *computation types* to equational logic over the standard simply typed lambda calculus with unit type 1, product types $\sigma \times \sigma'$ and function types $\sigma \rightarrow \sigma'$.

In the calculus, the unique (up to provable equality) element of the unit type 1 will be denoted $\langle\,\rangle$. First and second projection from a product type will be denoted by *fst* and *snd*, and pairing denoted by $\langle \_, \_ \rangle$; surjective pairing axioms form part of the equational logic. Typed lambda abstraction will be denoted by $\lambda x{:}\sigma.F(x)$, and application by MM'; beta and eta conversion axioms form part of the equational logic. We omit further details and concentrate instead on the rules for computation types.

To formulate these, we will be a little more precise about the allowed forms of judgement in the computational lambda calculus, which are

$$\begin{aligned} x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n &\vdash M : \sigma \\ x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n &\vdash M = M' : \sigma \end{aligned}$$

The intended meaning of these judgements is 'M is a term of type σ, given that the variables $x_1, \ldots, x_n$ have types $\sigma_1, \ldots, \sigma_n$ respectively' and 'M and M' are equal terms of type σ, given that the variables $x_1, \ldots, x_n$ have types $\sigma_1, \ldots, \sigma_n$ respectively'.

Remark 2.0.1 It will be a derived property of the systems we consider that for a given set of typing assumptions, a term has at most one type. Consequently we can abbreviate the second form of judgement to

$$x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash M = M'$$

without ambiguity.

The finite list on the left-hand side of the '$\vdash$' symbol in the above judgements will be called a *context* and typically abbreviated to Γ. Only judgements satisfying the well-formedness conditions

- the variables $x_1, \ldots, x_n$ are distinct
- the variables occurring in M lie in the list $x_1, \ldots, x_n$

will be considered. In particular, in giving rule schemes for generating judgements we assume that both the hypotheses and conclusion are well-formed—this obviates the need for side-conditions in certain rules. We will denote by $\Gamma, x{:}\sigma$ a context Γ extended by assigning type σ to a variable x not occurring in Γ; similarly Γ, Γ' denotes juxtaposed contexts with disjoint sets of variables. We omit the standard rules relating to the structure of contexts and the usual properties of equality.

2.1 Computation types

If σ is a type, so is $T\sigma$. The term-forming rules are

$$\textbf{values} \quad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash [M] : T\sigma}$$

$$\textbf{sequential composition} \quad \frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash F(x) : T\sigma'}{\Gamma \vdash \mathit{let}\, x {\Leftarrow} E \,\mathit{in}\, F(x) : T\sigma'}$$

Remark 2.1.1 Using the metatheoretical conventions mentioned in the Introduction, we should really write $let\, x{\Leftarrow}E\, in\, F(x)$ as $let(E, F)$—in other words, *let* is a meta-constant of type $\mathrm{EXP}{\rightarrow}(\mathrm{EXP}{\rightarrow}\mathrm{EXP}){\rightarrow}\mathrm{EXP}$. More trivially, a similar remark applies to value terms which, formally, make use of a meta-contant of type $\mathrm{EXP}{\rightarrow}\mathrm{EXP}$.

We think of terms of type $T\sigma$ as the (denotations of) 'computations of elements of type σ'. The intended meaning of the the value term $[M]$ is the trivial computation: 'immediately return the value M'. The intended meaning of the term $let\, x{\Leftarrow}E\, in\, F(x)$ is the following form of sequential composition of computations: 'first evaluate E, to x say, and then evaluate $F(x)$'. The reader may verify that the following equality rules respect these informal interpretations:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x{:}\sigma \vdash F(x) : T\sigma'}{\Gamma \vdash let\, x{\Leftarrow}[M]\, in\, F(x) = F(M)} \qquad \frac{}{e{:}T\sigma \vdash let\, x{\Leftarrow}e\, in\, [x] = e}$$

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash F(x) : T\sigma' \quad \Gamma, x'{:}T\sigma' \vdash G(x') : T\sigma''}{\Gamma \vdash let\, x'{\Leftarrow}(let\, x{\Leftarrow}E\, in\, F(x))\, in\, G(x') = let\, x{\Leftarrow}E\, in\, (let\, x'{\Leftarrow}F(x)\, in\, G(x'))}$$

This completes our review of the computational lambda calculus. Note that it is a *higher-order* calculus not only in the usual sense of permitting the formation of functionals of higher types (iterating the function-type constructor), but also because it permits the formation of computations of computations, and so on (iterating the computation-type constructor).

2.2 Categorical models

The interpretation in a cartesian closed category of the unit, product and function types and their associated terms and equations is well known: see Lambek and Scott [8], for example. Here we merely recall the overall shape of this interpretation in one particular formulation. Using finite products and exponentials in the cartesian closed category $\mathcal{C}$, an object $[\![\sigma]\!]$ of $\mathcal{C}$ is assigned to each type σ by induction on its structure. Similarly, by induction on the structure of terms M, each derivable typing judgement $\Gamma \vdash M : \sigma$ gives rise to a morphism in $\mathcal{C}$

$$[\![\Gamma \vdash M : \sigma]\!] : [\![\Gamma]\!] \longrightarrow [\![\sigma]\!]$$

whose domain is by definition the finite product

$$[\![\Gamma]\!] \overset{\text{def}}{=} [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$$

when the context Γ is $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$. The interpretation satisfies all provable equalities, in the sense that if $\Gamma \vdash M = M' : \sigma$ is derivable then $[\![\Gamma \vdash M : \sigma]\!]$ and $[\![\Gamma \vdash M' : \sigma]\!]$ are equal morphisms in $\mathcal{C}$.

Moggi has shown that to interpret computation types and their associated terms and equations in $\mathcal{C}$, one needs to give the extra structure of a *strong monad* on $\mathcal{C}$. This is a functor $T : \mathcal{C} \longrightarrow \mathcal{C}$ equipped with natural transformations

$$\begin{array}{rcll} \eta_X & : & X \longrightarrow T(X) & \textbf{(unit)} \\ \mu_X & : & T(T(X)) \longrightarrow T(X) & \textbf{(multiplication)} \\ t_{X,Y} & : & X \times T(Y) \longrightarrow T(X \times Y) & \textbf{(strength)} \end{array}$$

satisfying a number of commutative diagrams (seven, in fact)—see [11] for the full definition in this form. An equivalent definition, which is both simpler and closer to the syntax of computation types, is that of 'indexed Kleisli triple'. This is specified by a map on objects, sending X to $T(X)$, together with the following data

unit: for each object X, a morphism $\eta_X : X \longrightarrow T(X)$

lifting: for each morphism $f : X \times Y \longrightarrow T(Z)$, a morphism

$$f^* : X \times T(Y) \longrightarrow T(Z)$$

satisfying the following conditions, the first of which expresses naturality of lifting in the parameter X and the rest of which are parameterized versions of the usual axioms for a Kleisli triple (see [11, Definition 1.2]).

- given $f : X \longrightarrow X'$ and $g : X' \times Y \longrightarrow T(Z)$, then
$$(g \circ (f \times id_Y))^* = g^* \circ (f \times id_{T(Y)})$$
- given $f : X \times Y \longrightarrow T(Z)$, then $f^* \circ (id_X \times \eta_Y) = f$
- $(\eta_Y \circ \pi_2)^* = \pi_2 : X \times T(Y) \longrightarrow T(Y)$
- given $f : X \times Y \longrightarrow T(Z)$ and $g : X \times Z \longrightarrow T(W)$, then
$$(g^* \circ \langle \pi_1, f \rangle)^* = g^* \circ \langle \pi_1, f^* \rangle$$

(where the π_i are appropriate projection morphisms and $\langle , \rangle$ denotes pairing of morphisms).

Given such a structure on $\mathcal{C}$, we extend the interpretation of types as objects of $\mathcal{C}$ by defining

$$[\![T\sigma]\!] \stackrel{\text{def}}{=} T([\![\sigma]\!])$$

and extend the interpretation of terms by defining

$$[\![\Gamma \vdash [M] : T\sigma]\!] \stackrel{\text{def}}{=} \eta_{[\![\sigma]\!]} \circ [\![\Gamma \vdash M : \sigma]\!]$$

and

$$[\![\Gamma \vdash \mathit{let}\, x {\Leftarrow} E \,\mathit{in}\, F(x) : T\sigma']\!] \stackrel{\text{def}}{=} f^* \circ \langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash E : T\sigma]\!] \rangle$$

where f is $[\![\Gamma, x{:}\sigma \vdash F(x) : T\sigma']\!] : [\![\Gamma]\!] \times [\![\sigma]\!] \longrightarrow T[\![\sigma']\!]$. The conditions on unit and lifting given above ensure that this interpretation satisfies the equalities in 2.1.

3 Evaluation modalities

The informal interpretation of the value and sequential composition terms was given in Section 2.1 in terms of a relation of evaluation between computations and values (of a given type). In the computational lambda calculus, this intended meaning is only indirectly captured through its equational consequences. We are now going to embed the computational lambda calculus in a typed predicate logic containing not only atomic formulas for equality at each type, but also formulas expressing evaluation of computations to values. In fact evaluation will not be an atomic formula, but rather will be derived from 'modal quantifiers' which are really the key feature of the logic. The logic also contains propositional connectives: in this paper we will only consider conjunction and (intuitionistic) disjunction.

3.1 Formulas

To introduce the allowed formulas, we will give rules for deriving jugements of the form

$$\Gamma \vdash \phi \text{ prop}$$

where as in the previous section, $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ is a context assigning types to variables. To be well-formed, the above judgement must satisfy the condition that the variables occurring in ϕ lie in Γ. The intended meaning of the judgement is of course that 'ϕ is a well-formed proposition, given that the variables $x_1, \ldots, x_n$ have types $\sigma_1, \ldots, \sigma_n$ respectively'. The rules for deriving such judgements are:

Equality

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M' : \sigma}{\Gamma \vdash M = M' \text{ prop}}$$

Finite conjunction

$$\frac{}{\vdash \mathit{true} \text{ prop}} \qquad \frac{\Gamma \vdash \phi \text{ prop} \quad \Gamma \vdash \psi \text{ prop}}{\Gamma \vdash \phi \wedge \psi \text{ prop}}$$

Finite disjunction

$$\frac{}{\vdash \mathit{false} \text{ prop}} \qquad \frac{\Gamma \vdash \phi \text{ prop} \quad \Gamma \vdash \psi \text{ prop}}{\Gamma \vdash \phi \vee \psi \text{ prop}}$$

Evaluation modalities

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \mathrm{prop}}{\Gamma \vdash [x{\Leftarrow}E]\phi(x)\ \mathrm{prop}} \qquad \frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \mathrm{prop}}{\Gamma \vdash \langle x{\Leftarrow}E\rangle\phi(x)\ \mathrm{prop}}$$

The intended meaning of $[x{\Leftarrow}E]\phi(x)$ is: 'if E evaluates to x, then necessarily $\phi(x)$ holds'. The intended meaning of $\langle x{\Leftarrow}E\rangle\phi(x)$ is: 'it is possible for E to evaluate to an x for which $\phi(x)$ holds'.

Remark 3.1.1 Just as in Remark 2.1.1, we note that the above notation is an informal one which has been adopted for readability. Thus $[x{\Leftarrow}E]\phi(x)$ and $\langle x{\Leftarrow}E\rangle\phi(x)$ stand for $\Box(E,\phi)$ and $\Diamond(E,\phi)$ respectively, where $\Box$ and $\Diamond$ are meta-constants of the higher type $\mathrm{EXP}{\to}(\mathrm{EXP}{\to}\mathrm{EXP}){\to}\mathrm{EXP}$.

3.2 Entailment

To specify the logical properties of the above formulas, we will give rules for deriving judgements of the form

$$\Gamma, \Phi \vdash \psi$$

where Γ is a context (as defined in Section 2), Φ is a finite set of formulas, ψ is a formula, and the variables occurring in Φ and ψ lie in Γ. The intended meaning of the judgement is an intuitionistic sequent asserting 'ψ is logically entailed by the hypotheses Φ'. As usual, if Φ is empty, a singleton $\{\phi\}$, or a union $\Phi_1 \cup \Phi_2$, we write $\Gamma, \Phi \vdash \psi$ as

$$\Gamma \vdash \psi\ , \qquad \Gamma, \phi \vdash \psi \quad \text{and} \quad \Gamma, \Phi_1, \Phi_2 \vdash \psi$$

respectively. Finally, we will write

$$\Gamma, \phi \dashv\vdash \psi$$

to indicate that both $\Gamma, \phi \vdash \psi$ and $\Gamma, \psi \vdash \phi$ are derivable.

The rules concerning the logical properties of equality, conjunction and disjunction are the standard rules for this fragment of intuitionistic predicate calculus (see Dummett [3]). Note that with the conventions mentioned in the previous paragraph, the equality judgement

$$\Gamma \vdash M = M'$$

used in Section 2 is now taken as the particular instance of the entailment judgement with no hypothesis formulas and conclusion formula $M = M'$. So we can use the rules of the computational lambda calculus concerning product, function and computation types to derive entailment judgements. Finally, the rules concerning evaluation modalities are as follows.

3.2.1 Evaluation modalities preserve entailment:

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma, \phi(x) \vdash \psi(x)}{\Gamma, [x{\Leftarrow}E]\phi(x) \vdash [x{\Leftarrow}E]\psi(x)} \qquad \frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma, \phi(x) \vdash \psi(x)}{\Gamma, \langle x{\Leftarrow}E\rangle\phi(x) \vdash \langle x{\Leftarrow}E\rangle\psi(x)}$$

3.2.2 Values:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \text{prop}}{\Gamma, \phi(M) \dashv\vdash [x{\Leftarrow}[M]]\phi(x)} \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \text{prop}}{\Gamma, \phi(M) \dashv\vdash \langle x{\Leftarrow}[M]\rangle\phi(x)}$$

3.2.3 Sequential composition:

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash F(x) : T\sigma' \quad \Gamma, x'{:}T\sigma' \vdash \psi(x')\ \text{prop}}{\Gamma, [x{\Leftarrow}E][x'{\Leftarrow}F(x)]\psi(x') \dashv\vdash [x'{\Leftarrow}(\mathit{let}\, x{\Leftarrow}E\ \mathit{in}\, F(x))]\psi(x')}$$

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash F(x) : T\sigma' \quad \Gamma, x'{:}T\sigma' \vdash \psi(x')\ \text{prop}}{\Gamma, \langle x{\Leftarrow}E\rangle\langle x'{\Leftarrow}F(x)\rangle\psi(x') \dashv\vdash \langle x'{\Leftarrow}(\mathit{let}\, x{\Leftarrow}E\ \mathit{in}\, F(x))\rangle\psi(x')}$$

3.2.4 Necessity modality preserves finite conjunctions:

$$\frac{}{x{:}\sigma, e{:}T\sigma \vdash [x{\Leftarrow}e]\mathit{true}}$$

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \text{prop} \quad \Gamma, x{:}\sigma \vdash \psi(x)\ \text{prop}}{\Gamma, [x{\Leftarrow}E]\phi(x), [x{\Leftarrow}E]\psi(x) \vdash [x{\Leftarrow}E](\phi(x) \wedge \psi(x))}$$

3.2.5 Possibility modality preserves finite disjunctions:

$$\frac{}{x{:}\sigma, e{:}T\sigma, \langle x{\Leftarrow}e\rangle\mathit{false} \vdash \mathit{false}}$$

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \text{prop} \quad \Gamma, x{:}\sigma \vdash \psi(x)\ \text{prop}}{\Gamma, \langle x{\Leftarrow}E\rangle(\phi(x) \vee \psi(x)) \vdash \langle x{\Leftarrow}E\rangle\phi(x) \vee \langle x{\Leftarrow}E\rangle\psi(x)}$$

3.2.6 Possibility and necessity:

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \text{prop} \quad \Gamma, x{:}\sigma \vdash \psi(x)\ \text{prop}}{\Gamma, [x{\Leftarrow}E]\phi(x), \langle x{\Leftarrow}E\rangle\psi(x) \vdash \langle x{\Leftarrow}E\rangle(\phi(x) \wedge \psi(x))}$$

3.2.7 Possibility and equality:

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma \vdash N : \sigma' \quad \Gamma \vdash N' : \sigma' \quad \Gamma, x{:}\sigma \vdash \phi(x)\ \text{prop}}{\Gamma, N = N' \wedge \langle x{\Leftarrow}E\rangle\phi(x) \dashv\vdash \langle x{\Leftarrow}E\rangle(N = N' \wedge \phi(x))}$$

Remark 3.2.8 Using equality, truth and falsity, we get derived formulas asserting *evaluation* of computations to values, and *convergence* and *divergence* of computations:

$$\begin{array}{rcll} E \Rightarrow M & \stackrel{\text{def}}{=} & \langle x{\Leftarrow}E\rangle(x = M) & \text{`$E$ can evaluate to $M$'} \\ E{\Downarrow} & \stackrel{\text{def}}{=} & \langle x{\Leftarrow}E\rangle \mathit{true} & \text{`E can converge'} \\ E{\Uparrow} & \stackrel{\text{def}}{=} & [x{\Leftarrow}E]\mathit{false} & \text{`E must diverge'} \end{array}$$

One can also formulate partial and total correctness statements quite naturally in this language. Given a formula specifying an input-output relation from σ to σ'

$$\Gamma, x{:}\sigma, x'{:}\sigma' \vdash \psi(x, x') \text{ prop}$$

and a formula restricting the domain of admissible inputs

$$\Gamma, x{:}\sigma \vdash \phi(x) \text{ prop}$$

then we may say that a computation $\Gamma, x{:}\sigma \vdash F(x) : T\sigma'$ is partially correct for the specification if

$$\Gamma, x{:}\sigma, \phi(x) \vdash [x'{\Leftarrow}F(x)]\psi(x, x')$$

is derivable, and totally correct if

$$\Gamma, x{:}\sigma, \phi(x) \vdash [x'{\Leftarrow}F(x)]\psi(x, x') \wedge F(x){\Downarrow}$$

is derivable. Note that from rule 3.2.6, the conclusion of the second judgement entails $\langle x'{\Leftarrow}F(x)\rangle\psi(x, x')$.

We mention some simple consequences of the rules for evaluation modalities.

Proposition 3.2.9 *(i) Using the definition given in Remark 3.2.8 for the formula $E \Rightarrow M$, expressing evaluation of $E{:}T\sigma$ to $M{:}\sigma$, the following rules are derivable.*

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash [M] \Rightarrow M} \qquad \frac{\Gamma \vdash E \Rightarrow M \quad \Gamma \vdash F(E) \Rightarrow M'}{\Gamma \vdash (\mathit{let}\, x{\Leftarrow}E \,\mathit{in}\, F(x)) \Rightarrow M'}$$

Thus in the logic we indeed get a formalization of the intended behaviour mentioned in Section 2.1 of value and sequential composition terms under evaluation

(ii) The judgement

$$x{:}\sigma, x'{:}\sigma, [x] = [x'] \vdash x = x'$$

is derivable. This expresses the 'mono condition' on the unit of the strong monad T—see [11].

□

Remark 3.2.10 We indicate briefly the relation between our evaluation modalities and the propositional modal operators of existing program logics, which take the form

$$[P]\phi \qquad \text{and} \qquad \langle P\rangle\phi$$

with P a program and ϕ a proposition. (See Kozen and Tiuryn [7] for a survey.)

The first point is that *we can interpret (the denotation of) programs as computations of unit type, i.e. as terms of type* $T(1)$. Termination of the program corresponds to evaluation of $P{:}T(1)$ to the unique value $\langle\,\rangle{:}1$. (This idea can be seen in practice in the language Standard ML [10], which combines higher-order functional and imperative features: see Section 4.)

Secondly, since every term of type 1 is provably equal to $\langle\,\rangle$, specifying a formula $\phi(x)$ depending on a variable $x{:}1$ amounts to specifying a formula $\phi(\langle\,\rangle)$ depending upon no variables, i.e. a proposition. Given $P{:}T(1)$ and a proposition ϕ, the evaluation modalities yield new propositions $[x{\Leftarrow}P]\phi$ and $\langle x{\Leftarrow}P\rangle\phi$ which we abbreviate to $[P]\phi$ and $\langle P\rangle\phi$ respectively.

Specializing the rules given above for evaluation modalities to the case $\sigma = 1$, we obtain some standard properties of the program modalities, such as:

$$\begin{array}{rcl}
[P]\phi \wedge \langle P\rangle\psi & \vdash & \langle P\rangle(\phi \wedge \psi) \\
[\mathit{skip}]\phi & \dashv\vdash & \phi \\
\phi & \dashv\vdash & \langle \mathit{skip}\rangle\phi \\
[P;P']\phi & \dashv\vdash & [P][P']\phi \\
\langle P;P'\rangle & \dashv\vdash & \langle P\rangle\langle P'\rangle\phi
\end{array}$$

where *skip* stands for the value term $[\langle\,\rangle]$ and $P;P'$ for the sequential composition $\mathit{let}\, x{\Leftarrow}P \,\mathit{in}\, P'$ of two terms of type $T(1)$. Moreover, the equality rules for computation types in Section 2.1 imply associativity and unitary laws:

$$\begin{array}{ll}
\vdash & (P;P');P'' = P;(P';P'') \\
\vdash & \mathit{skip};P = P \\
\vdash & P;\mathit{skip} = P
\end{array}$$

3.3 Categorical models

In Section 2.2 we reviewed how Moggi's computational lambda calculus is interpreted in a cartesian closed category, $\mathcal{C}$, equipped with a strong monad, T. To extend the interpretation to the predicate logic described above, we use the standard technique of categorical logic originating with Lawvere [9] of interpreting formulas in a suitable 'hyperdoctrine' over $\mathcal{C}$. Since here we are only considering provability rather than proofs, it is sufficient to consider the case of hyperdoctrines which are *$\mathcal{C}$-indexed meet semilattices*, $\mathcal{P}$, equipped with suitable extra structure appropriate to the particular logic under consideration. Thus $\mathcal{P}$ is at least a contravariant functor on $\mathcal{C}$ valued in the category meet semilattices (the category whose objects are posets possessing meets of all finite subsets (including the empty meet), and whose morphisms are functions which preserve finite meets). In other words, for each object X of $\mathcal{C}$ one has a meet semilattice $\mathcal{P}(X)$ of 'properties' of X; and for each morphism $f : X \longrightarrow Y$ in $\mathcal{C}$ one has a function $\mathcal{P}(f) : \mathcal{P}(Y) \longrightarrow \mathcal{P}(X)$ of 'substitution along f' which preserves finite meets. The action of $\mathcal{P}(f)$ on $B \in \mathcal{P}(Y)$ will be written $f^{-1}B$. Contravariant functoriality of $\mathcal{P}$ means that one has $id^{-1}B = B$ and $g^{-1}(f^{-1}B) = (f \circ g)^{-1}B$.

Given such a $\mathcal{P}$, we wish to interpret each derivable judgement $\Gamma \vdash \phi$ prop as an element

$$[\![\Gamma \vdash \phi \text{ prop}]\!] \in \mathcal{P}([\![\Gamma]\!])$$

where as before, the object $[\![\Gamma]\!]$ in $\mathcal{C}$ is the finite product $[\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$ when Γ is the context $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$. We can then define the *satisfaction* of an entailment judgement $\Gamma, \Phi \vdash \psi$ to mean that

$$\bigwedge_{\phi \in \Phi} [\![\Gamma \vdash \phi \text{ prop}]\!] \leq [\![\Gamma \vdash \psi \text{ prop}]\!]$$

holds in the meet semilattice $\mathcal{P}([\![\Gamma]\!])$ (where $\bigwedge$ indicates finite meet and $\leq$ the partial order).

The definition of $[\![\Gamma \vdash \phi \text{ prop}]\!]$ proceeds by induction on the structure of the formula ϕ, and each of the predicate formers in the logic requires some appropriate properties or extra structure on $\mathcal{P}$ to interpret it soundly. Soundness means that we should be able to prove for each rule of inference that the conclusion is satisfied when all of the hypotheses are.

The conditions on $\mathcal{P}$ needed to interpret finite conjunctions and disjunctions are quite standard: each $\mathcal{P}(X)$ should be a distributive lattice and each $f^{-1} : \mathcal{P}(Y) \longrightarrow \mathcal{P}(X)$ should, in addition to preserving finite meets, also preserve the finite joins as well. The property of $\mathcal{P}$ needed to soundly interpret equality is almost as well-known, except that in the absence of propositional implication certain subtleties arise which would otherwise be masked. Briefly, we require all the order-preserving functions

$$(\Delta_X \times id_Y)^{-1} : \mathcal{P}((X \times X) \times Y) \longrightarrow \mathcal{P}(X \times Y) \tag{1}$$

(where Δ_X is the diagonal $\langle id_X, id_X \rangle$) to possess left adjoints, $\exists_{\Delta_X \times id_Y}$, and for these left adjoints to satisfy certain conditions termed by Lawvere [9] 'Beck-Chevalley' and 'Frobenius Reciprocity' conditions; moreover, equality of morphisms in $\mathcal{C}$ must be implied by satisfaction in $\mathcal{P}$ of the corresponding equality formula. Since we wish to concentrate here on describing the categorical semantics of the evaluation modalities, we merely refer the reader to [2] for more details.

Definition 3.3.1 (Cf. [12, Definition 4.8]) A *T-modality* on $\mathcal{P}$ is specified by a family of order-preserving functions

$$\Box_{X,Y} : \mathcal{P}(X \times Y) \longrightarrow \mathcal{P}(X \times T(Y))$$

one for each pair of objects X, Y in $\mathcal{C}$, satisfying the following three conditions relating to the formulation of the strong monad T as an 'indexed Kleisli triple' as in Section 2.2.

Naturality condition: given $f : X \longrightarrow X'$, then

$$\begin{tikzcd}
\mathcal{P}(X' \times Y) \arrow[r, "(f \times id_Y)^{-1}"] \arrow[d, "\Box_{X',Y}"'] & \mathcal{P}(X \times Y) \arrow[d, "\Box_{X,Y}"] \\
\mathcal{P}(X' \times T(Y)) \arrow[r, "(f \times id_{T(Y)})^{-1}"'] & \mathcal{P}(X \times T(Y))
\end{tikzcd}$$

commutes.

Unit condition: for all X and Y

$$\begin{tikzcd}
\mathcal{P}(X \times Y) \arrow[r, "\Box_{X,Y}"] \arrow[dr, "id"'] & \mathcal{P}(X \times T(Y)) \arrow[d, "(id_X \times \eta_Y)^{-1}"] \\
 & \mathcal{P}(X \times Y)
\end{tikzcd}$$

commutes.

Lifting condition: given $f : X \times Y \longrightarrow T(Z)$, then

$$\begin{tikzcd}
\mathcal{P}(X \times Z) \arrow[r, "\Box_{X,Z}"] \arrow[d, "\Box_{X,Z}"'] & \mathcal{P}(X \times T(Z)) \arrow[r, "\langle \pi_1, f \rangle^{-1}"] & \mathcal{P}(X \times Y) \arrow[d, "\Box_{X,Y}"] \\
\mathcal{P}(X \times T(Z)) \arrow[rr, "\langle \pi_1, f^* \rangle^{-1}"'] & & \mathcal{P}(X \times T(Y))
\end{tikzcd}$$

commutes.

Given two such T-modalities on $\mathcal{P}$, $\Box$ and $\Diamond$, we can interpret the evaluation modalities as follows. Suppose, inductively, that we have

$$\begin{aligned} [\![\Gamma]\!] &= X \\ [\![\sigma]\!] &= Y \\ [\![\Gamma \vdash E : \sigma]\!] &= f : X \longrightarrow T(Y) \\ [\![\Gamma, x{:}\sigma \vdash \phi(x) \text{ prop}]\!] &= A \in \mathcal{P}(X \times Y) \end{aligned}$$

Then we define

$$\begin{aligned} [\![\Gamma \vdash [x {\Leftarrow} E]\phi(x) \text{ prop}]\!] &\stackrel{\text{def}}{=} \langle id_X, f\rangle^{-1}(\Box_{X,Y} A) \\ [\![\Gamma \vdash \langle x {\Leftarrow} E\rangle\phi(x) \text{ prop}]\!] &\stackrel{\text{def}}{=} \langle id_X, f\rangle^{-1}(\Diamond_{X,Y} A) \end{aligned}$$

The conditions imposed on $\Box$ and $\Diamond$ by Definition 3.3.1 ensure that the rules in 3.2.1,3.2.2 and 3.2.3 are sound for this interpretation. To ensure soundness for the rules in 3.2.4, clearly we require

(i) each $\Box_{X,Y} : \mathcal{P}(X \times Y) \longrightarrow \mathcal{P}(X \times T(Y))$ preserves finite meets

whilst to ensure soundness for the rules in 3.2.5 we require

(ii) each $\Diamond_{X,Y} : \mathcal{P}(X \times Y) \longrightarrow \mathcal{P}(X \times T(Y))$ preserves finite joins.

Soundness for 3.2.6 requires

(iii) for all objects X, Y and all elements $A, B \in \mathcal{P}(X \times Y)$

$$\Diamond_{X,Y}(A) \wedge \Box_{X,Y}(B) \leq \Diamond_{X,Y}(A \wedge B)$$

Finally, soundness for 3.2.7 requires

(iv) for all objects X, Y and all elements $A, B \in \mathcal{P}(X \times Y)$

$$\exists_{\Delta_X \times id_{T(Y)}}(\Diamond_{X,Y} A) = \Diamond_{X \times X, Y}(\exists_{\Delta_X \times id_Y} A)$$

where $\exists_{\Delta_X \times id_Y}$ denotes the left adjoint to the function (1) mentioned above.

3.4 Examples

We give two examples of the categorical structure developed in the previous section. In fact, the underlying cartesian closed category in both examples is the same—namely the category of ω-cpo's (*without* least element) and ω-continuous functions, which we will denote by $\mathcal{C}po$. Thus the objects of $\mathcal{C}po$ are sets equipped with a partial order possessing least upper bounds of all countably infinite chains; the morphisms are functions preserving order and least upper bounds of countably infinite chains. Finite products in $\mathcal{C}po$ are created by the forgetful functor to the category of sets; an exponential $X \rightarrow Y$ in $\mathcal{C}po$ is given by the hom-set $\mathcal{C}po(X, Y)$ ordered pointwise from Y.

Of course $\mathcal{C}po$ (and its subcategories) has very much more structure than this, which is used in the traditional domain-theoretic approach to denotational semantics (see [4] for a survey). Moggi has noticed that the way in which this extra structure is actually used is via the construction of various strong monads reflecting various aspects of computation. From the many examples in [11, 13] we select those of *partiality* and *partiality with side-effects* and show how to extend them to models of our Evaluation Logic by giving suitable hyperdoctrines of properties equipped with modalities.

Example 3.4.1 (Partiality) For each ω-cpo X, recall that the *lifted* ω-cpo, $X_\perp$ is obtained by adjoining a least element to X. Specifically, to fix notation, define

$$X_\perp \overset{\text{def}}{=} \{[x] \mid x \in X\} \cup \{\perp\}$$

with partial order given by

> $e \sqsubseteq e'$ in $X_\perp$ iff for all $x \in X$, if $e = [x]$ then there is some $x' \in X$ with $e' = [x']$ and $x \sqsubseteq x'$ in X.

Then $X \mapsto X_\perp$ is the object part of a strong monad on $\mathcal{C}po$ whose unit functions $\eta_X : X \longrightarrow X_\perp$ are $x \mapsto [x]$, and whose lifting operation sends the ω-continuous function $f : X \times Y \longrightarrow Z_\perp$ to $f^* : X \times Y_\perp \longrightarrow Z_\perp$ where for all $x \in D$ and $e \in Y_\perp$

$$f^*(x, e) = \begin{cases} f(x, y) & \text{if } e = [y] \\ \perp & \text{if } e = \perp \end{cases}$$

In the resulting model of the computational lambda calculus, the only quality of computation modelled is non-termination ($\perp$). Computations of values in Y with a parameter in X, i.e. elements of the ω-cpo $X \rightarrow Y_\perp$, amount to partial continuous functions from X to Y.

To extend the model to formulas, we consider the $\mathcal{C}po$-indexed meet semilattice of *inclusive subsets*, $\mathcal{I}$. Its value, $\mathcal{I}(X)$, at an ω-cpo X is the set of all subsets $A \subseteq X$ which are closed under taking least upper bounds in X of countably infinite chains lying in A. For each ω-continuous function $f : X \longrightarrow Y$, $f^{-1} : \mathcal{I}(Y) \longrightarrow \mathcal{I}(X)$ sends an inclusive $B \in \mathcal{I}(Y)$ to the inverse image $f^{-1}B \overset{\text{def}}{=} \{x \in X \mid f(x) \in B\}$. Finite meets in $\mathcal{I}(X)$ are given by set-theoretic intersection.

It is not hard to see that the union of finitely many inclusive subsets of an ω-cpo is again inclusive. Consequently, each $\mathcal{I}(X)$ is a distributive lattice and each f^{-1} preserves finite joins in addition to finite meets. So $\mathcal{I}$ soundly models finite conjunction and disjunction. It also has the requisite properties to model equality formulas: this follows from the fact that the direct image of an inclusive $A \in \mathcal{I}(X \times Y)$ along the function $(x, y) \mapsto (x, x, y)$ is just $\{(x, x, y) \mid (x, y) \in A\}$, which is again inclusive. (It is not true in general that the image of an inclusive subset along an ω-continuous function is inclusive.)

Turning now to modelling the evaluation modalities, we define the following two modalities on $\mathcal{I}$ for the lifting monad:

$$\begin{array}{rcl} \Box_{X,Y}(A) & \stackrel{\text{def}}{=} & \{(x,e) \in X \times Y_\bot \mid \forall y \in Y \,.\, e = [y] \supset (x,y) \in A\} \\ \Diamond_{X,Y}(A) & \stackrel{\text{def}}{=} & \{(x,e) \in X \times Y_\bot \mid \exists y \in Y \,.\, e = [y] \wedge (x,y) \in A\} \end{array}$$

for all ω-cpo's X, Y and inclusive $A \in \mathcal{I}(X \times Y)$. The inclusivity of the right-hand sides in these definitions is not completely trivial, bearing in mind the fact that $\mathcal{I}$ is not a model of (intuitionistic) logic with either existential quantification or implication. Nevertheless the conditions in Definition 3.3.1 and conditions (i)–(iv) listed after that definition are all easily verified for these modalities. Indeed much more special properties hold of these particular modalities than we have required in our Evaluation Logic—namely $\Diamond_{X,Y}$ and $\Box_{X,Y}$ give respectively left and right adjoints to $(id_X \times \eta_Y)^{-1}$, and satisfy various Beck-Chevalley and Frobenius Reciprocity conditions. The categorical logic of this more special kind of modality is studied in [1, 2].

Example 3.4.2 (Side-effects) Let S be some fixed ω-cpo of 'states'. For example, if a state is completely specified by the contents of memory locations $\ell_1, \ell_2, \ldots$ each of which can contain a number, then S would be the set of finite partial functions from addresses to numbers (equipped with the discrete partial order of equality).

A computation of a value in X whose evaluation has some (unspecified) side-effects on the current state can be modelled in an extensional way by a (continuous) function $e : S \rightarrow (X \times S)_\bot$. Thus given any initial state $s \in S$, either $e(s) = \bot$, i.e. the computation does not terminate, or $e(s) = [(x, s')]$, i.e. the computation terminates yielding the value $x \in X$ and a new state $s' \in S$. Defining

$$T^S(X) \stackrel{\text{def}}{=} S \rightarrow (X \times S)_\bot$$

then $X \mapsto T^S(X)$ is the object part of a strong monad on $\mathcal{C}po$ whose unit functions are given by

$$\eta_X(x)(s) = [(x, s)]$$

and whose lifting operation sends $f : X \times Y \longrightarrow T^S(Z)$ to the function $f^* : X \times T^S(Y) \longrightarrow T^S(Z)$ given by

$$f^*(x,e)(s) = \begin{cases} f(x,y)(s') & \text{if } e(s) = [(y,s')] \\ \bot & \text{if } e(s) = \bot \end{cases}$$

Over $\mathcal{C}po$ equipped with this stong monad T^S we consider the following indexed meet semilattice, $\mathcal{I}^S$. The value of $\mathcal{I}^S$ at an ω-cpo X is the collection of inclusive subsets of the product $X \times S$, partially ordered by inclusion. In other words

$$\mathcal{I}^S(X) \stackrel{\mathrm{def}}{=} \mathcal{I}(X \times S)$$

with $\mathcal{I}$ as in the previous example. Similarly, given $f : X \longrightarrow Y$ in $\mathcal{C}po$, then $f^{-1} : \mathcal{I}^S(Y) \longrightarrow \mathcal{I}^S(X)$ is defined to be the inverse image function $(f \times id_S)^{-1}$. We will adopt the following notation when referring to elements of $\mathcal{I}^S(X)$: given $A \in \mathcal{I}^S(X)$, $s \in S$ and $x \in X$, we write

$$s \Vdash A(x)$$

to indicate that $(x, s) \in A$, and say 'in state s, x has property A'.

It follows immediately from the definition of $\mathcal{I}^S$ in terms of $\mathcal{I}$ that $\mathcal{I}^S$ inherits from the latter the properties needed to model equality formulas and finite conjunctions and disjunctions.

Turning now to modelling the evaluation modalities, we give two T^S-modalities, $\Box^S$ and $\Diamond^S$, on $\mathcal{I}^S$. For each $A \in \mathcal{I}^S(X \times Y)$, define $\Box^S_{X,Y}A$ and $\Diamond^S_{X,Y}A$ in $\mathcal{I}^S(X \times T^S(Y))$ by declaring that for all $s \in S$, $x \in X$ and $e \in T^S(Y)$

$$\begin{array}{lll} s \Vdash (\Box^S_{X,Y}A)(x,e) & \text{iff} & \forall s' \in S \,.\, \forall y \in Y \,.\, e(s) = [(y,s')] \supset s' \Vdash A(x,y) \\ s \Vdash (\Diamond^S_{X,Y}A)(x,e) & \text{iff} & \exists s' \in S \,.\, \exists y \in Y \,.\, e(s) = [(y,s')] \wedge s' \Vdash A(x,y) \end{array}$$

It is not hard to verify that these definitions do indeed yield inclusive subsets—either directly, or by noting that they can be expressed in terms of the modalities of Example 3.4.1:

$$\begin{array}{rcl} \Box^S_{X,Y}(A) & = & \{(x,e,s) \mid (x, e(s)) \in \Box_{X,Y\times S}(A)\} \\ \Diamond^S_{X,Y}(A) & = & \{(x,e,s) \mid (x, e(s)) \in \Diamond_{X,Y\times S}(A)\} \end{array}$$

Indeed this observation serves to show that $\Box^S$ and $\Diamond^S$ inherit from the lifting-modalities $\Box$ and $\Diamond$ the properties (i)–(iv) given after Definition 3.3.1. Finally it can be verified directly from the definition of the strong monad T^S that both $\Box^S$ and $\Diamond^S$ satisfy the naturality, unit and lifting conditions required by Definition 3.3.1.

4 Translating Natural Semantics

In this section we will give an example of translating programming language features into suitable *theories* over Evaluation Logic. As discussed in the Introduction, the aim is to see how well fitted is this logic for capturing operational behaviour specified in terms of the 'Natural Semantics' style of operational semantics [18, 5].

An Evaluation Logic theory is specified by a *signature* of type- and term-constructors (and possibly formula-constructors, although we will not need these here), together with the *axioms* of the theory—which are a collection of judgements (involving expressions generated over the given signature). We will present such a theory for a fragment of the Standard ML language [10] containing both functional features (higher-order recursive function declarations) and imperative features (assignable global variables), and which we call Tiny-ML. We begin by specifying the programming language syntax.

4.1 Tiny-ML **Types**

These are the simple types over a ground types of integers, `int`, and a one-element ground type, `unit`:

$$\sigma ::= \mathtt{int} \mid \mathtt{unit} \mid \sigma_1 {\to} \sigma_2$$

4.2 Tiny-ML **Expressions**

These are given by the following grammar:

$$\begin{array}{lcll}
\mathtt{e} & ::= & x & \text{variables}\\
& \mid & \mathtt{()} & \text{unit value}\\
& \mid & \mathtt{n} & \text{integer values}\\
& \mid & \mathtt{op}(\mathtt{e}_1, \mathtt{e}_2) & \text{arithmetic operations}\\
& \mid & \mathtt{if}\,\mathtt{e}_1 = \mathtt{0}\,\mathtt{then}\,\mathtt{e}_2\,\mathtt{else}\,\mathtt{e}_3 & \text{conditionals}\\
& \mid & \mathtt{fn}\,x{:}\sigma \Rightarrow \mathtt{e} & \lambda\text{-abstractions}\\
& \mid & \mathtt{e}_1\mathtt{e}_2 & \text{applications}\\
& \mid & \mathtt{letrec}\,f(x) = \mathtt{e}_1{:}\sigma\ \mathtt{in}\ f(\mathtt{e}_2)\ \mathtt{end} & \text{recursive functions[1]}\\
& \mid & \ell := \mathtt{e} & \text{assignment to } \ell\\
& \mid & !\ell & \text{contents of } \ell
\end{array}$$

Here x ranges over a countably infinite set of variables, `n` ranges over the integers, $\mathtt{op} \in \{+, -, *\}$, and ℓ ranges over a countably infinite set of global memory locations (constants of type `int ref`, in ML parlance). Recalling from the Introduction the metatheoretical treatment of variable-binding, we remark that the form $\mathtt{fn}\,x{:}\sigma \Rightarrow \mathtt{e}$ is in fact long-hand for the meta-expression $\mathtt{fn}(\sigma, (x)\mathtt{e})$ with `fn` a meta-constant of type $\textsc{exp}{\to}(\textsc{exp}{\to}\textsc{exp}){\to}\textsc{exp}$. Similarly, $\mathtt{letrec}\,f(x) = \mathtt{e}_1{:}\sigma\ \mathtt{in}\ f(\mathtt{e}_2)\ \mathtt{end}$ is long-hand for $\mathtt{letrec}((f)(x)\mathtt{e}_1, \sigma, \mathtt{e}_2)$ with `letrec` a meta-constant of type $(\textsc{exp}{\to}\textsc{exp}{\to}\textsc{exp}){\to}\textsc{exp}{\to}\textsc{exp}{\to}\textsc{exp}$.

Perhaps Tiny-ML is not so tiny. Here are some derived forms of expression:

$$\begin{array}{rcl}
\mathtt{let}\,x = \mathtt{e}_1{:}\sigma\ \mathtt{in}\,\mathtt{e}_2 & \overset{\text{def}}{=} & (\mathtt{fn}\,x{:}\sigma \Rightarrow \mathtt{e}_2)\mathtt{e}_1\\
\mathtt{e}_1; \mathtt{e}_2 & \overset{\text{def}}{=} & \mathtt{let}\,x = \mathtt{e}_1{:}\mathtt{unit}\ \mathtt{in}\,\mathtt{e}_2\\
\mathtt{while}\,\mathtt{e}_1 \neq \mathtt{0}\ \mathtt{do}\,\mathtt{e}_2 & \overset{\text{def}}{=} & \mathtt{letrec}\,f(x) =\\
& & \quad (\mathtt{if}\,\mathtt{e}_1 = \mathtt{0}\,\mathtt{then}\,\mathtt{()}\,\mathtt{else}\,\mathtt{e}_2; x){:}\mathtt{unit}\ \mathtt{in}\ f(\mathtt{()})\ \mathtt{end}
\end{array}$$

[1] This should be regarded as an abbreviation for the legal Standard ML expression $\mathtt{let}\,\mathtt{val}\,\mathtt{rec}\,f = \mathtt{fn}\,x \Rightarrow \mathtt{e}_1{:}\sigma\ \mathtt{in}\ f(\mathtt{e}_2)\ \mathtt{end}$.

4.3 Static Semantics of Tiny-ML

Table 1 gives the rules for deriving type assignments, $\Gamma \vdash \mathsf{e} : \sigma$, giving the Tiny-ML type σ of a Tiny-ML expression e in a context $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$. Conventions about contexts are the same as those in Section 2 for the computational lambda calculus.

Remark 4.3.1 It is easy to see that *in a given context a* Tiny-ML *expression has at most one type*: if $\Gamma \vdash \mathsf{e} : \sigma$ and $\Gamma \vdash \mathsf{e} : \sigma'$ are both derivable from the rules in Table 1 then $\sigma \equiv \sigma'$.

4.4 Dynamic Semantics of Tiny-ML

4.4.1 Closed and canonical expressions

Let us call a Tiny-ML expression e *closed* if there is some type σ for which $\vdash \mathsf{e} : \sigma$ is derivable. Thus e contains no (free) variables, and by Remark 4.3.1 σ is uniquely determined by e.

Then the *canonical* Tiny-ML expressions, c, comprise the subset of all closed Tiny-ML expressions given by the following grammar:

$$\mathsf{c} ::= \texttt{()} \mid \texttt{n} \mid \texttt{fn}\, x{:}\sigma \Rightarrow \mathsf{e}$$

4.4.2 States

These are the finite functions, s, from the set of global memory locations to the set of integers. If s is a state with domain $\{\ell_1, \ldots, \ell_k\}$ and with $s(\ell_i) = n_i$, we will write s as

$$s = \{\ell_1 \mapsto n_1, \ldots, \ell_k \mapsto n_k\}$$

We write

$$s(\ell \mapsto n)$$

for the *updated* state which maps ℓ to n and otherwise acts like s.

4.4.3 The evaluation relation

This is a relation of the form

$$s, \mathsf{e} \twoheadrightarrow s', \mathsf{c}$$

where s and s' are states, e is a closed Tiny-ML expression and c is a canonical one. The relation is inductively defined by the rules given in Table 2. In stating these rules we use the following simplifying convention from [10]:

'Sequentiality' convention. A rule scheme of the form

$$\frac{\mathsf{e}_1 \twoheadrightarrow \mathsf{c}_1 \quad \mathsf{e}_2 \twoheadrightarrow \mathsf{c}_2 \quad \cdots \quad \mathsf{e}_n \twoheadrightarrow \mathsf{c}_n}{\mathsf{e} \twoheadrightarrow \mathsf{c}}$$

$$\frac{}{\Gamma, x{:}\sigma, \Gamma' \vdash x : \sigma}$$

$$\frac{}{\Gamma \vdash \mathtt{()} : \mathtt{unit}} \qquad \frac{}{\Gamma \vdash \mathtt{n} : \mathtt{int}}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{int} \quad \Gamma \vdash \mathtt{e}_2 : \mathtt{int}}{\Gamma \vdash \mathtt{op}(\mathtt{e}_1, \mathtt{e}_2) : \mathtt{int}}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{int} \quad \Gamma \vdash \mathtt{e}_2 : \sigma \quad \Gamma \vdash \mathtt{e}_3 : \sigma}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_1 = \mathtt{0}\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \sigma}$$

$$\frac{\Gamma, x{:}\sigma \vdash \mathtt{e} : \sigma'}{\Gamma \vdash \mathtt{fn}\ x{:}\sigma \Rightarrow \mathtt{e} : \sigma \rightarrow \sigma'} \qquad \frac{\Gamma \vdash \mathtt{e} : \sigma \rightarrow \sigma' \quad \Gamma \vdash \mathtt{e}' : \sigma}{\Gamma \vdash \mathtt{e}\mathtt{e}' : \sigma'}$$

$$\frac{\Gamma, f{:}\sigma \rightarrow \sigma', x{:}\sigma \vdash \mathtt{e}_1 : \sigma' \quad \Gamma \vdash \mathtt{e}_2 : \sigma}{\Gamma \vdash \mathtt{letrec}\ f(x) = \mathtt{e}_1{:}\sigma'\ \mathtt{in}\ f(\mathtt{e}_2)\ \mathtt{end} : \sigma'}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{int}}{\Gamma \vdash \ell := \mathtt{e} : \mathtt{unit}} \qquad \frac{}{\vdash !\ell : \mathtt{int}}$$

Table 1: Rules for the static semantics of Tiny-ML

is an abbreviation for the rule scheme

$$\frac{s_1, \mathsf{e}_1 \twoheadrightarrow s_2, \mathsf{c}_1 \quad s_2, \mathsf{e}_2 \twoheadrightarrow s_3, \mathsf{c}_2 \quad \cdots \quad s_n, \mathsf{e}_n \twoheadrightarrow s_{n+1}, \mathsf{c}_n}{s_1, \mathsf{e} \twoheadrightarrow s_{n+1}, \mathsf{c}}$$

(Thus the order of hypotheses is significant when using this convention.)

Remark 4.4.1 It is not hard to see from the form of the rules in Table 2 that if $s, \mathsf{e} \twoheadrightarrow s', \mathsf{c}$ is derivable, then s and s' have the same domain, and e and c have the same type.

4.5 Translation of TINY-ML into Evaluation Logic

We begin by giving the theory over Evaluation Logic into which we will translate TINY-ML. As we said above, such a theory is specified by a signature and a collection of axioms.

4.5.1 Signature

The signature of the theory has a single ground type Z (type of integers) and meta-constants

$$\begin{array}{rcl}
n & : & \mathrm{EXP} \\
op & : & \mathrm{EXP}\to\mathrm{EXP}\to\mathrm{EXP} \\
cond & : & \mathrm{EXP}\to\mathrm{EXP}\to\mathrm{EXP}\to\mathrm{EXP} \\
rec_\sigma & : & (\mathrm{EXP}\to\mathrm{EXP}\to\mathrm{EXP})\to\mathrm{EXP}\to\mathrm{EXP} \\
up_\ell & : & \mathrm{EXP}\to\mathrm{EXP} \\
ct_\ell & : & \mathrm{EXP}
\end{array}$$

where n ranges over the set of integers, op ranges over $\{+, -, *\}$, ℓ ranges over the set of global memory locations, and σ ranges over the set of types of the computational lambda calculus (see Section 2) generated from the ground type Z. The rules for introducing these meta-constants are given in Table 3. Note the difference between the last three rules in this table and the corresponding last three rules in Table 1.

4.5.2 Axioms

The intended meaning of the term $cond(M_1, M_2, M_3){:}\sigma$ is a conditional branching on whether $M_1{:}Z$ is equal to *0* or not. The intended meaning of the term $rec_{\sigma'}(E', M){:}T\sigma'$ is the computation of the value at $M{:}\sigma$ of the function recursively defined by the declaration

$$f(x) \overset{\mathrm{def}}{=} E'(f, x)$$

$$\frac{}{\mathtt{c} \twoheadrightarrow \mathtt{c}}\ (\mathtt{c}\ \text{canonical})$$

$$\frac{\mathtt{e}_1 \twoheadrightarrow \mathtt{0} \quad \mathtt{e}_2 \twoheadrightarrow \mathtt{c}}{\mathtt{if}\ \mathtt{e}_1 = \mathtt{0}\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \twoheadrightarrow \mathtt{c}} \qquad \frac{\mathtt{e}_1 \twoheadrightarrow \mathtt{n} \quad \mathtt{e}_3 \twoheadrightarrow \mathtt{c}}{\mathtt{if}\ \mathtt{e}_1 = \mathtt{0}\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \twoheadrightarrow \mathtt{c}}\ (\text{if } n \neq 0)$$

$$\frac{\mathtt{e}_1 \twoheadrightarrow \mathtt{n}_1 \quad \mathtt{e}_2 \twoheadrightarrow \mathtt{n}_2}{\mathtt{op}(\mathtt{e}_1, \mathtt{e}_2) \twoheadrightarrow \mathtt{n}}\ (\text{if } op(n_1, n_2) = n)$$

$$\frac{\mathtt{e}_1 \twoheadrightarrow \mathtt{fn}\ x{:}\sigma \Rightarrow \mathtt{e} \quad \mathtt{e}_2 \twoheadrightarrow \mathtt{c} \quad \mathtt{e}(\mathtt{c}/x) \twoheadrightarrow \mathtt{c}'}{\mathtt{e}_1\mathtt{e}_2 \twoheadrightarrow \mathtt{c}'}$$

$$\frac{\mathtt{e}_2 \twoheadrightarrow \mathtt{c} \quad \mathtt{e}_1((\mathtt{fn}\ x{:}\sigma \Rightarrow \mathtt{letrec}\ f(x') = \mathtt{e}_1{:}\sigma'\ \mathtt{in}\ f(x)\ \mathtt{end})/f\,,\ \mathtt{c}/x) \twoheadrightarrow \mathtt{c}'}{\mathtt{letrec}\ f(x) = \mathtt{e}_1{:}\sigma'\ \mathtt{in}\ f(\mathtt{e}_2)\ \mathtt{end} \twoheadrightarrow \mathtt{c}'}$$

$$\frac{s, \mathtt{e} \twoheadrightarrow s', \mathtt{n}}{s, \ell := \mathtt{e} \twoheadrightarrow s'(\ell \mapsto n), \mathtt{()}} \qquad \frac{}{s, !\ell \twoheadrightarrow s, \mathtt{n}}\ (\text{if } n = s(\ell))$$

Table 2: Rules for the dynamic semantics of Tiny-ML

(The type subscript on $rec_{\sigma'}$ is there to retain the uniqueness of type property mentioned in Remark 2.0.1.) The intended meaning of the term $up_\ell(M){:}T1$ is the 'program' (i.e. computation of unit type—cf. Remark 3.2.10) which assigns value $M{:}Z$ to location ℓ. The intended meaning of the term $ct_\ell{:}TZ$ is the computation which just returns the contents of location ℓ.

With these intended interpretations in mind, Table 4 formalizes certain expected properties: these are the axioms of the particular Evaluation Logic theory we wish to consider, and which are added to the rules for generating entailment judgements given in Sections 2 and 3. Some of the axioms employ a derived form of sequential composition

$$E; E' \overset{\text{def}}{=} \mathit{let}\, x {\Leftarrow} E \;\mathit{in}\, E'$$

in which the second computation is independent of the parameter x. Thus the derived rule of formation is

$$\frac{\Gamma \vdash E : T\sigma \quad \Gamma \vdash E' : T\sigma'}{\Gamma \vdash E; E' : T\sigma'}$$

and from the equational properties of *let* given in Section 2.1 one can derive

$$\begin{array}{rcl} e{:}T\sigma, e'{:}T\sigma', e''{:}T\sigma'' & \vdash & (e; e'); e'' = e; (e'; e'') \\ e{:}T\sigma & \vdash & \mathit{skip}; e = e \\ e{:}T1 & \vdash & e; \mathit{skip} = e \end{array}$$

where by definition, *skip* is $[\langle\,\rangle]$.

4.5.3 Translation

The basic idea is that TINY-ML expressions $\mathsf{e}{:}\sigma$ are translated into terms of *computation* type, $[\![\mathsf{e}]\!]{:}T[\![\sigma]\!]$, in the computational lambda calculus. Furthermore, since the dynamic semantics of TINY-ML is strict ('call-by-value'), if e depends on some variables $x_1{:}\sigma_1, \ldots, x{:}\sigma_n$, then $[\![\mathsf{e}]\!]$ should depend on variables ranging over the values of the translated types, $x_1{:}[\![\sigma_1]\!], \ldots, x_n{:}[\![\sigma_n]\!]$ (rather than on variables ranging over computations). Thus at the heart of the translation is Moggi's [11] call-by-value translation of lambda calculus into his computational lambda calculus (see also Plotkin [16]).

The translation is defined by induction on the structure of TINY-ML types and expressions by the clauses in Table 5. Note that for a canonical TINY-ML expression c (as defined in Section 4.4.1), the translation takes the form

$$[\![\mathsf{c}]\!] = [|\mathsf{c}|]$$

(where $|\mathsf{c}|$ is of type $[\![\sigma]\!]$ when c is of type σ). Thus $|\mathsf{c}|$ is the translation of the canonical term c as a *value* rather than as a computation.

4.6 Adequacy of the translation

The translation given by Table 5 is adequate for both the static and dynamic semantics of Tiny-ML. For the static semantics this means

Proposition 4.6.1 (Static Adequacy) $x_1{:}\sigma_1, \dots, x_n{:}\sigma_n \vdash \mathsf{e} : \sigma$ *is derivable from the rules in Table 1 if and only if*

$$x_1{:}[\![\sigma_1]\!], \dots, x_n{:}[\![\sigma_n]\!] \vdash [\![\mathsf{e}]\!] : T[\![\sigma]\!]$$

is derivable from the type assignment rules of the computational lambda calculus augmented by the rules in Table 3.

The triviality of the type system of Tiny-ML makes this proposition easy to prove by induction on the structure of e.

For the dynamic semantics, we first have to translate the operational evaluation relation

$$s, \mathsf{e} \twoheadrightarrow s', \mathsf{c} \tag{2}$$

into a corresponding judgement in Evaluation Logic. Taking into account Remark 4.4.1, we may suppose that e and c have the same type and that s and s' have equal domain. Supposing the states are

$$\begin{array}{rcl} s & = & \{\ell_1 \mapsto n_1, \dots, \ell_k \mapsto n_k\} \\ s' & = & \{\ell_1 \mapsto n'_1, \dots, \ell_k \mapsto n'_k\} \end{array}$$

then we will translate this instance of the evaluation relation into the Evaluation Logic judgement

$$\vdash \langle \vec{up}(\vec{n})\rangle \langle x \Leftarrow [\![e]\!]\rangle \langle \vec{x'} \Leftarrow \vec{ct}\rangle \left(x = |\mathsf{c}| \ \wedge \ \bigwedge_{i=1}^{k} x'_i = n'_i \right) \tag{3}$$

Here $\langle \vec{up}(\vec{n})\rangle$ and $\langle \vec{x'} \Leftarrow \vec{ct}\rangle$ stand for iterated modalities, viz

$$\begin{array}{rcl} \langle \vec{up}(\vec{n})\rangle & \stackrel{\text{def}}{=} & \langle up_{\ell_1}(n_1)\rangle \cdots \langle up_{\ell_k}(n_k)\rangle \\ \langle \vec{x'} \Leftarrow \vec{ct}\rangle & \stackrel{\text{def}}{=} & \langle x'_1 \Leftarrow ct_{\ell_1}\rangle \cdots \langle x'_k \Leftarrow ct_{\ell_k}\rangle \end{array}$$

where we are using the abbreviation mentioned in Remark 3.2.10 to write $\langle x \Leftarrow E\rangle\phi$ as

$$\langle E\rangle\phi$$

when E is a term of type $T1$ (such as $up_{\ell_i}(n_i)$) and ϕ does not depend upon x.

We hope the reader will agree that (modulo the unfamiliar formalism) the judgement (3) is a *natural* rendering of the operational evaluation relation into our logic, since it says something like: 'it is possible to make the assignments to $\ell_1, \dots, \ell_k$ to create the state s, then possible to evaluate $[\![\mathsf{e}]\!]$ to a value equal to $|\mathsf{c}|$ and have those locations contain the values of state s' as a result'. In any case, one can prove

Proposition 4.6.2 (Dynamic Adequacy) *If the evaluation relation* (2) *is derivable from the rules in Table* 2*, then the corresponding jugdgement* (3) *is derivable in Evaluation Logic from the theory described by Tables* 3 *and* 4*. The converse holds when* `c` *is of ground type (*`int` *or* `unit`*).*

Proof The proof of the first sentence is by induction on the derivation of (2). For the second sentence we use the fact that Example 3.4.2 yields a *model* of the Evaluation Logic theory we are considering. Since it is a model, derivability of (3) in the logic implies its satisfaction in the model. Assuming `e` is of (ground) type `gnd`, satisfaction in this model amount to requiring that

$$[\![\mathsf{e}]\!](s) = (|\mathsf{c}|, s') \in T^S([\![\mathtt{gnd}]\!])$$

where now $[\![\_]\!]$ is essentially the standard domain-theoretic semantics of Tiny-ML (see Mosses [14] for example)—from which it is known that we can recover the operational relation (2).

□

Concluding remarks

Evaluation Logic, we would claim, is a good medium in which to formulate logical principles reflecting the kind of operational behaviour expressible in Natural Semantics. The Tiny-ML example we have given here is certainly too simple to really test this claim. However, note that even here the logic allows us to reason about the behaviour of expressions-with-state without having to specify a global state explicitly—unlike the traditional domain-theoretic approach (and its formalizations). This becomes much more important for forms of computation where a domain-theoretic modelling of global state is very complicated (or not known). Computation involving dynamically allocated resources is an example of this, and an appropriate Evaluation Logic is currently under development. (Of course, one still has to find concrete models of the logical theories which arise ...)

Another aspect of the over-simplicity of the Tiny-ML example is that it is in fact possible to eliminate the use of evaluation modalities and give a version of the above 'dynamic adequacy' result purely within an equational theory over the computational lambda calculus. Indeed one can equate the evaluation relation (2) with satisfaction of the equation

$$\vdash (up_{\ell_1}(n_1); \cdots ; up_{\ell_k}(n_k); [\![\mathsf{e}]\!]) = (up_{\ell_1}(n'_1); \cdots ; up_{\ell_k}(n'_k); [\![\mathsf{c}]\!])$$

in the theory we have given in Table 4, minus the last two axioms. However, the full modal logic should come into its own when devising computationally adequate theories for languages with non-deterministic features, for example. Even for purely deterministic languages, evaluation modalities appear useful when we go beyond simple computational adequacy results and address the question of finding logical principles for reasoning about the behaviour of programs in all (observable) contexts. The rules of Evaluation Logic and the axioms in Table 4 are *more* than adequate for Proposition 4.6.2 (not all of them are used in its proof), but are not exhaustive for reasoning about observable equivalence (since the latter is not recursively axiomatizable). An interesting example of the need for the evaluation modalities can be found in [1, 2], where the necessity modality is used to express an induction principle for fixpoint computations.

References

[1] R. L. Crole and A. M. Pitts, *New Foundations for Fixpoint Computations*, Proc. 5th Annual Symposium on Logic in Computer Science, Philadelphia (IEEE Computer Society Press, Washington, 1990) 489–497.

[2] R. L. Crole and A. M. Pitts, *New Foundations for Fixpoint Computations: FIX-Hyperdoctrines and the FIX-Logic*, University of Cambridge Computer Laboratory Technical Report No. 204, August 1990.

[3] M. Dummett, *Elements of Intuitionism* (Oxford University Press, 1977).

[4] C. Gunter and D. S. Scott, *Semantic Domains.* Chapter in *Handbook of Theoretical Computer Science* (North-Holland, Amsterdam, 1990).

[5] G. Kahn, *Natural Semantics.* In K. Fuchi and M. Nivat (eds), *Programming of Future Generation Computers* (Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1988) 237–258.

[6] J. W. Klop, *Combinatory Reduction Systems*, Amsterdam Mathematical Center Tracts 129 (1980).

[7] D. Kozen and J. Tiuryn, *Logics of Programs.* Chapter in *Handbook of Theoretical Computer Science* (North-Holland, Amsterdam, 1990).

[8] J. Lambek and P. J. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge Studies in Advanced Mathematics 7 (Cambridge University Press, 1986).

[9] F. W. Lawvere, *Equality in Hyperdoctrines and the Comprehension Schema as an Adjoint Functor.* In A. Heller (ed.), *Applications of Categorical Algebra* (Amer. Math. Soc., Providence RI, 1970) 1–14.

[10] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML* (The MIT Press, Cambridge Massachussetts, 1990).

[11] E. Moggi, *Computational lambda-calculus and monads*, Proc. 4th Annual Symposium on Logic in Computer Science, Asilomar CA (IEEE Computer Society Press, Washington, 1989) 14–23.

[12] E. Moggi, *Notions of Computations and Monads*, preprint, 1989.

[13] E. Moggi, Lecture notes on *An Abstract View of Programming Languages*, July 1989.

[14] P. D. Mosses, *Denotational Semantics.* Chapter in *Handbook of Theoretical Computer Science* (North-Holland, Amsterdam, 1990).

[15] B. Nordström, K. Petersson and J. M. Smith, *Programming in Martin-Löf's Type Theory, An Introduction* (Oxford University Press, 1990).

[16] G. D. Plotkin, *Call-by-Name, Call-by-Value and the λ-Calculus*, Theoretical Computer Science 1(1977) 125–159.

[17] G. D. Plotkin, *LCF considered as a programming language*, Theoretical Computer Science 5(1977) 223–255.

[18] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Aarhus University Computer Science Department Report DAIMI FN-19, 1981.

[19] G. D. Plotkin, *Denotational semantics with partial functions*, unpublished lecture notes from CSLI Summer School, 1985.

[20] D. S. Scott, *A type-theoretic alternative to CUCH, ISWIM, OWHY*, unpublished manuscript, University of Oxford, 1969.

[21] R. A. G. Seely, *Hyperdoctrines, Natural Deduction and the Beck Condition*, Zeitschr. f. math. Logik und Grundlagen d. Math. 29 (1983) 505–542.

$$\frac{}{\vdash n : Z} \qquad \frac{\Gamma \vdash M : Z \quad \Gamma \vdash M' : Z}{\Gamma \vdash op(M, M') : Z}$$

$$\frac{\Gamma \vdash M_1 : Z \quad \Gamma \vdash M_2 : \sigma \quad \Gamma \vdash M_3 : \sigma}{\Gamma \vdash cond(M_1, M_2, M_3) : \sigma}$$

$$\frac{\Gamma, f{:}\sigma \rightarrow T\sigma', x{:}\sigma \vdash E'(f, x) : T\sigma' \quad \Gamma \vdash M : \sigma}{\Gamma \vdash rec_{\sigma'}(E', M) : T\sigma'}$$

$$\frac{\Gamma \vdash M : Z}{\Gamma \vdash up_\ell(M) : T1} \qquad \frac{}{\vdash ct_\ell : TZ}$$

Table 3: The signature of the theory

$$\frac{}{x{:}\sigma, x'{:}\sigma \vdash \mathit{cond}(0, x, x') = x}$$

$$\frac{}{x{:}\sigma, x'{:}\sigma \vdash \mathit{cond}(n, x, x') = x'} \ (\text{if } n \neq 0)$$

$$\frac{}{\vdash \mathit{op}(n_1, n_2) = n} \ (\text{if } \mathit{op}(n_1, n_2) = n)$$

$$\frac{\Gamma, f{:}\sigma \rightarrow T\sigma', x{:}\sigma \vdash E'(f, x) : T\sigma'}{\Gamma, x{:}\sigma \vdash \mathit{rec}_{\sigma'}(E', x) = E'(\lambda x{:}\sigma.\mathit{rec}_{\sigma'}(E', x)\, ,\, x)}$$

$$\frac{}{x{:}Z, x'{:}Z \vdash \mathit{up}_\ell(x); \mathit{up}_\ell(x') = \mathit{up}_\ell(x')}$$

$$\frac{}{x{:}Z, x'{:}Z \vdash \mathit{up}_\ell(x); \mathit{up}_{\ell'}(x') = \mathit{up}_{\ell'}(x'); \mathit{up}_\ell(x)} \ (\text{if } \ell \neq \ell')$$

$$\frac{}{x{:}Z \vdash \mathit{up}_\ell(x); \mathit{ct}_\ell = \mathit{up}_\ell(x); [x]}$$

$$\frac{}{x{:}Z \vdash \mathit{up}_\ell(x); \mathit{ct}_{\ell'} = \mathit{let}\, x' \Leftarrow \mathit{ct}_{\ell'}\, \mathit{in}\, (\mathit{up}_\ell(x); [x'])} \ (\text{if } \ell \neq \ell')$$

$$\frac{}{\vdash \mathit{let}\, x \Leftarrow \mathit{ct}_\ell\, \mathit{in}\, (\mathit{let}\, x' \Leftarrow \mathit{ct}_{\ell'}\, \mathit{in}\, [\langle x, x' \rangle]) = \mathit{let}\, x' \Leftarrow \mathit{ct}_{\ell'}\, \mathit{in}\, (\mathit{let}\, x \Leftarrow \mathit{ct}_\ell\, \mathit{in}\, [\langle x, x' \rangle])}$$

$$\frac{}{x{:}Z \vdash \mathit{up}_\ell(x)\Downarrow}$$

Table 4: Theory axioms

Types

$$
\begin{aligned}
\llbracket \mathtt{int} \rrbracket &\overset{\mathrm{def}}{=} Z \\
\llbracket \mathtt{unit} \rrbracket &\overset{\mathrm{def}}{=} 1 \\
\llbracket \sigma \to \sigma' \rrbracket &\overset{\mathrm{def}}{=} \llbracket \sigma \rrbracket \to T \llbracket \sigma' \rrbracket
\end{aligned}
$$

Expressions

$$
\begin{aligned}
\llbracket x \rrbracket &\overset{\mathrm{def}}{=} [x] \\
\llbracket \mathtt{()} \rrbracket &\overset{\mathrm{def}}{=} [\langle\, \rangle] \\
\llbracket \mathtt{n} \rrbracket &\overset{\mathrm{def}}{=} [n] \\
\llbracket \mathtt{op}(\mathtt{e}_1, \mathtt{e}_2) \rrbracket &\overset{\mathrm{def}}{=} \mathit{let}\, x_1 \Leftarrow \llbracket \mathtt{e}_1 \rrbracket\ \mathit{in} \\
&\qquad (\mathit{let}\, x_2 \Leftarrow \llbracket \mathtt{e}_2 \rrbracket\ \mathit{in}\, [\mathit{op}(x_1, x_2)]) \\
\llbracket \mathtt{if}\ \mathtt{e}_1 = \mathtt{0}\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \rrbracket &\overset{\mathrm{def}}{=} \mathit{let}\, x \Leftarrow \llbracket \mathtt{e}_1 \rrbracket\ \mathit{in}\, \mathit{cond}(x, \llbracket \mathtt{e}_2 \rrbracket, \llbracket \mathtt{e}_3 \rrbracket) \\
\llbracket \mathtt{fn}\, x{:}\sigma \Rightarrow \mathtt{e} \rrbracket &\overset{\mathrm{def}}{=} [\lambda x{:}\llbracket \sigma \rrbracket . \llbracket \mathtt{e} \rrbracket] \\
\llbracket \mathtt{e}_1 \mathtt{e}_2 \rrbracket &\overset{\mathrm{def}}{=} \mathit{let}\, f \Leftarrow \llbracket \mathtt{e}_1 \rrbracket\ \mathit{in}\, (\mathit{let}\, x \Leftarrow \llbracket \mathtt{e}_2 \rrbracket\ \mathit{in}\, f x) \\
\llbracket \mathtt{letrec}\, f(x) = \mathtt{e}_1{:}\sigma'\ \mathtt{in}\ f(\mathtt{e}_2)\ \mathtt{end} \rrbracket &\overset{\mathrm{def}}{=} \mathit{let}\, x' \Leftarrow \llbracket \mathtt{e}_2 \rrbracket\ \mathit{in}\ \mathit{rec}_{\llbracket \sigma' \rrbracket}((f)(x) \llbracket \mathtt{e}_1 \rrbracket\, , x') \\
\llbracket \ell := \mathtt{e} \rrbracket &\overset{\mathrm{def}}{=} \mathit{let}\, x \Leftarrow \llbracket \mathtt{e} \rrbracket\ \mathit{in}\ \mathit{up}_\ell(x) \\
\llbracket !\ell \rrbracket &\overset{\mathrm{def}}{=} \mathit{ct}_\ell
\end{aligned}
$$

Table 5: Translation of Tiny-ML types and expressions