A fully abstract semantics for a higher-order functional language with nondeterministic computation

ALAN JEFFREY

ABSTRACT. This paper is about the relationship between the theory of *monadic types* and the practice of *concurrent functional programming*. We present a typed functional programming language *CMML*, with a type system based on Moggi's monadic metalanguage, and concurrency based on Reppy's Concurrent ML. We present an operational and denotational semantics for the language, and show that the denotational semantics is fully abstract for may-testing. We show that a fragment of CML can be translated into CMML, and that the translation is correct up to weak bisimulation.

Contents

Alan Jeffrey
School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH, UK
alanje@cogs.susx.ac.uk

1 Introduction

This paper shows how the notion of computation types interacts with operational and denotational semantics for a nondeterministic λ-calculus.

In a conventional call-by-value typed λ-calculus, one can add a nondeterminism operator $e \sqcap f$ with typing:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash f : \tau}{\Gamma \vdash e \sqcap f : \tau}$$

and provide it with nondeterministic reductions:

$$e \sqcap f \Longrightarrow e \qquad e \sqcap f \Longrightarrow f$$

However, such a λ-calculus does not fit the usual 'off the shelf' (Lambek and Scott, 1986) categorical model of cartesian closed categories (cccs), since it does not satisfy either η- or β-equivalence. For example:

$$(\lambda x\,.\,(x,x))(0 \sqcap 1) \neq (0 \sqcap 1, 0 \sqcap 1)$$

since the latter has the reduction:

$$(0 \sqcap 1, 0 \sqcap 1) \Longrightarrow (0,1)$$

which the former cannot match. Similarly:

$$((\lambda x\,.\,0) \sqcap (\lambda x\,.\,1)) \neq \lambda y\,.\,(((\lambda x\,.\,0) \sqcap (\lambda x\,.\,1))y)$$

since the latter (placed in an appropriate context) has the reduction:

$$(\lambda z\,.\,(z0,z0))(\lambda y\,.\,(((\lambda x\,.\,0) \sqcap (\lambda x\,.\,1))y)) \Longrightarrow (0,1)$$

which the former cannot match.

Since such a nondeterministic λ-calculus cannot be modelled as a ccc, the traditional denotational approach is to model it using a *powerdomain* functor (Plotkin, 1981), for example giving the semantics of integers as:

$$[\![\mathsf{int}]\!] = \mathcal{P}(\mathbf{N}_\perp)$$

Moggi (1991) observed that the phenomenon of non-trivial computation is quite general, and that the denotational semantics can be simplified by separating the semantics of computation (in this case the functor $\mathcal{P}(\_{}_\perp)$) from the semantics of data (in this case **N**). This separation can be achieved in the type system of the λ-calculus by providing a *computation type constructor* $\mathsf{C}\_$ whose semantics is given by an appropriate functor. For example, in the above case we have:

$$[\![\mathsf{int}]\!] = \mathbf{N} \qquad [\![\mathsf{C}\,\tau]\!] = \mathcal{P}([\![\tau]\!]_\perp)$$

In this treatment, we give different types to values such as '$2 : \mathsf{int}$' and '$1 + 1 : \mathsf{C}\,\mathsf{int}$'. The former is an integer *value* where the latter is an integer *computation*. This separation of expressions into values and computations of values is standard in the call-by-value λ-calculus, but is usually done syntactically rather than in the type system.

Computation types have had some success in the functional programming community in modelling systems with side-effects (Wadler, 1990), such as the Haskell (Hudak *et al.*, 1992) monadic I/O library (Gordon *et al.*, 1994).

In Section 3 we present a λ-calculus with an explicit type constructor, and show (assuming the programs satisfy certain equivalences) that its models are precisely given by categorical structures:

Programming construct	*Categorical equivalent*
Algebraic datatypes	Categories with finite products
Let-expressions	Strong monads
Functions	T-exponentials
If-then-else expressions	Computational coproducts of 1
Deconstructors	Morphisms in the Kleisli category

This gives us quite a powerful tool for giving semantics for languages with computation type: given such a language, we just have to verify (for example using operational techniques such as bisimulation) that it satisfies certain equivalences, from which we get 'for free' a canonical semantics in any category with the appropriate structure.

We use this technique in Section 4 to show how a fully abstract semantics can be given for the case of a nondeterministic language with recursion. The denotational semantics is given in the domain of algebraic dcpos, *not necessarily with least elements*. For example, we can compare the denotation of booleans with computations of booleans:

$$[\![\mathsf{bool}]\!] = (t \quad f) \qquad [\![\mathsf{C}\,\mathsf{bool}]\!] = \begin{pmatrix} & \{t,f\} & \\ \nearrow & & \nwarrow \\ \{t\} & & \{f\} \\ \nwarrow & & \nearrow \\ & \{\} & \end{pmatrix}$$

Since we are not requiring all types to have least elements, this gives a very natural semantics for data, using the product and coproduct structure of posets. However, we still need to give a denotation for fixed points, but the restrictive type system ensures that we only have to find fixed points of terms of computation type, and those always have least elements.

We can show that the denotational semantics is fully abstract for the operational semantics using a variant of Abramsky (1989) and Ong's (1988) *lazy lambda-calculus* and Abramsky's (1991) *domain theory in logical form*. This is similar to Ong's (1993) use of a program logic for the untyped λ-calculus, but is simplified by the fact that nondeterminism can only occur at computation type.

The simplified proof of full abstraction is due to the fact that the nondeterministic λ-calculus with computation types has more expressive power than the λ-calculus without. For example, in the nondeterministic λ-calculus, the following terms are identified:

$$(0 \sqcap 1, 0 \sqcap 1) = (0,0) \sqcap (0,1) \sqcap (1,0) \sqcap (1,1)$$

whereas their simplistic translations into the λ-calculus with computation types are not equal:

$$[([0] \sqcap [1], [0] \sqcap [1])] \neq [([0],[0])] \sqcap [([0],[1])] \sqcap [([1],[0])] \sqcap [([1],[1])]$$

since when placed in the context:

$$\mathsf{let}\, x \Leftarrow \_\, \mathsf{in}\, \mathsf{let}\, y \Leftarrow x.\mathrm{L}\, \mathsf{in}\, \mathsf{let}\, z \Leftarrow x.\mathrm{L}\, \mathsf{in}\, y = z$$

the former has the reduction:

$$\begin{aligned}&\mathsf{let}\, x \Leftarrow [([0] \sqcap [1], [0] \sqcap [1])]\, \mathsf{in}\, \mathsf{let}\, y \Leftarrow x.\mathrm{L}\, \mathsf{in}\, \mathsf{let}\, z \Leftarrow x.\mathrm{L}\, \mathsf{in}\, y = z\\ &\quad \Longrightarrow \mathsf{let}\, y \Leftarrow [0] \sqcap [1]\, \mathsf{in}\, \mathsf{let}\, z \Leftarrow [0] \sqcap [1]\, \mathsf{in}\, y = z\\ &\quad \Longrightarrow 0 = 1\\ &\quad \Longrightarrow [\mathsf{false}]\end{aligned}$$

which the latter cannot match.

This paper is part of an investigation into the use of computation types in concurrent functional languages (Jeffrey, 1995). There, the nondeterministic language is extended with communication capabilities based on Reppy's (1991, 1992) Concurrent ML, and we show that it can be given a fully abstract semantics based on Hennessy's (1994) fully abstract semantics for untyped higher-order processes. The resulting program logic has much of the flavour of Hennessy–Milner (1980) logic.

2 Mathematical preliminaries

This section contains the standard definitions and results which will be used throughout this paper.

2.1 Categories and monads

This section contains a brief overview of the categorical structure used in later sections.

We refer the reader to Pierce's (1991) introductory textbook or Mac Lane's (1971) book for the definitions of *category*, *functor*, *natural transform*, *product*, *coproduct*, *initial*, *terminal*, *isomorphic*, *equivalent*, and for further details on the definitions in this section.

Write $\mathbf{C}[X,Y]$ for the class of morphisms with source X and target Y in the category $\mathbf{C}$. When this class is a set, we call this a *homset*.

A *punctuated* category is one where the initial and terminal object coincide.

A *small* category is one where the class of objects and the class of arrows are sets.

Let **Set** be the category of sets with functions.

Let **Mon** be the category of monoids with monoid homomorphisms.

Let **Cat** be the category of small categories with functors.

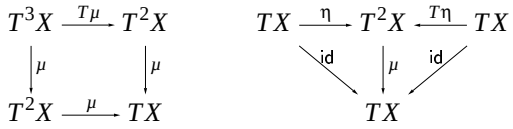
Let **CCat** be the category of small categories with distinguished finite products, and functors which preserve the product structure. We shall associate products to the left, writing $X_1 \times \cdots \times X_{n+1}$ for $(X_1 \times \cdots X_n) \times X_{n+1}$ and $X_1 \times \cdots \times X_0$ for 1. We shall similarly associate the mediating morphism $\langle f_1, \ldots, f_n \rangle : X \to X_1 \times \cdots \times X_n$ for $f_i : X \to X_i$ to the left, writing $\langle f_1, \ldots, f_{n+1} \rangle$ for $\langle \langle f_1, \ldots, f_n \rangle, f_{n+1} \rangle$ and $\langle f_1, \ldots, f_0 \rangle$ for !. We shall write $\pi : X \times Y \to X$ and $\pi' : X \times Y \to Y$ for the projections, and write $\pi_{m,n} : X_1 \times \cdots \times X_m \to X_n$ for the generalized projection.

Write $\mathbf{C}^{\mathsf{op}}$ for the *dual* category to $\mathbf{C}$, with objects from $\mathbf{C}$ and morphisms $f : Y \to X$ for each $f : X \to Y$ in $\mathbf{C}$. If $F : \mathbf{C} \to \mathbf{C}'$ is a functor, then so is $F^{\mathsf{op}} : \mathbf{C}^{\mathsf{op}} \to \mathbf{C}'^{\mathsf{op}}$ where $F^{\mathsf{op}} X = FX$ and $F^{\mathsf{op}} f = Ff$.

A *monad* is a functor $T : \mathbf{C} \to \mathbf{C}$ together with natural transformations:

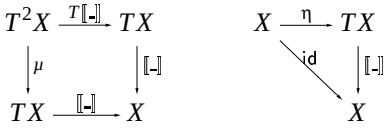
$$\eta_X : X \to TX \qquad \mu_X : T^2 X \to TX$$

such that:

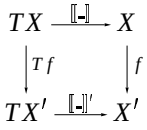


Note that here, and throughout this document, we omit subscripts where they can be deduced from context.

A T*-algebra* is a object X from **C** together with a morphism $[\![\_]\!] : TX \to X$ such that:



A T*-algebra morphism* between T-algebras $(X, [\![\_]\!])$ and $(X', [\![\_]\!]')$ is a morphism $f : X \to X'$ such that:

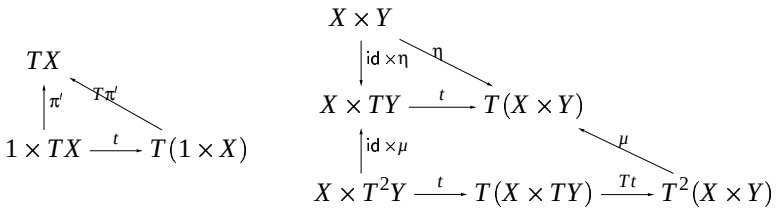


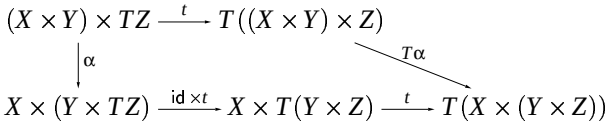
Let T-**Alg** be the category of T-algebras with T-algebra morphisms.

A monad on a category with finite products is *strong* iff it has a natural transform:

$$t_{X,Y} : X \times TY \to T(X \times Y)$$

such that:





where α is the *associativity* natural transformation:

$$\alpha : (X \times Y) \times Z \to X \times (Y \times Z)$$

Let **SMon** be the category of strong monads with functors which preserve the product and monad structure.

A *computational cartesian closed category* (*cccc*) is a category with finite products and a strong monad $T : \mathbf{C} \to \mathbf{C}$ such that for any objects X and Y there is a *T-exponential* object TX^Y with bijection (natural in X and Z):

$$\mathsf{curry} : \mathbf{C}[X \times Y, TZ] \simeq \mathbf{C}[X, TZ^Y]$$

Given a cccc, we can define the evaluation morphism as:

$$\mathsf{ev} = \mathsf{curry}^{-1}\mathsf{id} : TZ^Y \times Y \to TZ$$

Let **CCCC** be the category of small cccc's with functors which preserve the product, monadic and T-exponential structure.

2.2 Partial orders

This section contains a brief overview of the order structure used in later sections.

We refer the reader to Davey and Priestly's (1990) introductory textbook or Plotkin's (1981) lecture notes for the definitions of *poset*, *join*, *meet*, *monotone*, and for further details on the definitions in this section.

A poset $(X, \leq)$ is *discrete* iff $x \leq y$ implies $x = y$.

A subset Y of a poset X is *directed* iff every finite subset of Y has an upper bound in Y; note in particular that $\emptyset$ is *not* directed, but that any non-empty chain is. A *directed-complete partial order* (*dcpo*) is a poset where every directed set has a join. A function between dcpos is *continuous* iff it is monotone and respects directed join. Let **DCPO** be the category of dcpos and continuous functions. The *well-below* or *approximation* relation in a dcpo is defined:

$$x \ll y \text{ iff } y \leq \bigvee Z \Rightarrow \exists z \in Z \,.\, x \leq z \text{ for all directed } Z$$

An element x is *compact* iff $x \ll x$. Let $\mathop{\downarrow}\!Y = \{x \mid x \ll y \in Y\}$. A dcpo is *algebraic* iff:

$$\mathop{\downarrow}\!\{y\} \text{ is directed and } y = \bigvee \mathop{\downarrow}\!\{y\} \text{ for all } y$$

For any dcpo X, dcpo with least element Y, $x \in X$ and $y \in Y$, let $x \Rightarrow y : X \to Y$ be the *step function*:

$$(x \Rightarrow y) = x' \mapsto \begin{cases} y & \text{if } x \ll x' \\ \bot & \text{otherwise} \end{cases}$$

Let **Alg** be the category of algebraic dcpos with continuous functions. For any category **C** of posets, define the subcategories:

- $\mathbf{C}_\bot$ the subcategory of **C** of posets with a least element $\bot$, and morphisms which respect $\bot$.
- $\mathbf{C}_\vee$ the subcategory of **C** of posets with binary join $\vee$, and morphisms which respect $\vee$.

A category $\mathbf{C}$ is **DCPO**-*enriched* (resp. $\mathbf{DCPO}_\perp$-*enriched*) iff for every objects X and Y, the homset $\mathbf{C}[X,Y]$ forms a dcpo (resp. dcpo with $\perp$), and composition is continuous (resp. strict continuous). For example **Alg** is **DCPO**-enriched, and if $\mathbf{C}$ and $\mathbf{C}_i$ are **DCPO**-enriched, then so are $\mathbf{C}_\perp$, $\mathbf{C}_\vee$, $\mathbf{C}^{\mathsf{op}}$ and $\prod_i \mathbf{C}_i$. Similarly $\mathbf{Alg}_\perp$ is $\mathbf{DCPO}_\perp$-enriched, and if $\mathbf{C}$ and $\mathbf{C}_i$ are $\mathbf{DCPO}_\perp$-enriched, then so are $\mathbf{C}_\perp$, $\mathbf{C}_\vee$, $\mathbf{C}^{\mathsf{op}}$ and $\prod_i \mathbf{C}_i$.

A functor $F : \mathbf{C} \to \mathbf{C}'$ between **DCPO**-enriched categories is *locally monotone* (resp. *locally continuous*) iff its restriction to homsets $F : \mathbf{C}[X,Y] \to \mathbf{C}'[FX,FY]$ is monotone (resp. continuous).

For example, the following functors are locally continuous:

$$\begin{aligned}
(_ \to _) &: \mathbf{Alg}^{\mathsf{op}} \times \mathbf{Alg}_{\perp\vee} \to \mathbf{Alg}_{\perp\vee} \\
(_ \to_\vee _) &: \mathbf{Alg}^{\mathsf{op}}_\vee \times \mathbf{Alg}_{\perp\vee} \to \mathbf{Alg}_{\perp\vee} \\
(_ \to_{\perp\vee} _) &: \mathbf{Alg}^{\mathsf{op}}_{\perp\vee} \times \mathbf{Alg}_{\perp\vee} \to \mathbf{Alg}_{\perp\vee} \\
(_ \times _) &: \mathbf{Alg} \times \mathbf{Alg} \to \mathbf{Alg} \\
_\perp &: \mathbf{Alg}_\vee \to \mathbf{Alg}_{\perp\vee} \\
\mathcal{P} &: \mathbf{Alg} \to \mathbf{Alg}_{\perp\vee}
\end{aligned}$$

given by:

$$\begin{aligned}
X \to Y &= \mathbf{Alg}[X,Y] & f \to g &= h \mapsto f;h;g \\
X \to_\vee Y &= \mathbf{Alg}_\vee[X,Y] & f \to_\vee g &= h \mapsto f;h;g \\
X \to_{\perp\vee} Y &= \mathbf{Alg}_{\perp\vee}[X,Y] & f \to_{\perp\vee} g &= h \mapsto f;h;g \\
X \times Y &= \{(x,y) \mid x \in X, y \in Y\} & f \times g &= (x,y) \mapsto (fx, fy) \\
X_\perp &= \{\perp\} \cup \{\mathsf{lift}\, x \mid x \in X\} & f_\perp &= (\perp \mapsto \perp) \cup (\mathsf{lift}\, x \mapsto \mathsf{lift}\, fx) \\
\mathcal{P}X &= \{\textstyle\bigdownarrow Y \mid Y \subseteq X\} & \mathcal{P}f &= Y \mapsto \textstyle\bigdownarrow \{fy \mid y \in Y\}
\end{aligned}$$

where $X \times Y$ inherits the order from X and Y, $X_\perp$ inherits the order from X with new least element $\perp$, and $\mathcal{P}X$ is ordered by subset inclusion. In addition, if $F : \mathbf{C} \to \mathbf{C}'$, $G : \mathbf{C}' \to \mathbf{C}''$ and $H_i : \mathbf{C} \to \mathbf{C}_i$ are locally continuous, then so are F^{op}, $F;G$ and $\langle H_i \mid i \in I \rangle$.

An *embedding* in a $\mathbf{DCPO}_\perp$-enriched category $\mathbf{C}$ is a morphism $e : X \to Y$ such that there exists a morphism $e^R : Y \to X$ where $e;e^R = \mathsf{id}$ and $e^R;e \leq \mathsf{id}$. Let $\mathbf{C}_E$ be the subcategory of $\mathbf{C}$ where all morphisms are embeddings. Note that any locally monotone functor $F : \mathbf{C} \to \mathbf{C}'$ restricts to a functor $F : \mathbf{C}_E \to \mathbf{C}'_E$.

An ω-*diagram* in a catgegory $\mathbf{C}$ is a series of objects $X_1, X_2, \ldots$ with morphisms $f_{ij} : X_i \to X_j$ when $i \leq j$ such that $f_{ii} = \mathsf{id}$ and $f_{ij};f_{jk} = f_{ik}$. A *cocone* for such an ω-diagram is an object X with morphisms $f_i : X_i \to X$ such that $f_{ij};f_j = f_i$. A *colimit* is a cocone $f_i : X_i \to X$ such that for any other cocone $f'_i : X_i \to X'$ there is a unique $f : X \to X'$ such that $f_i;f = f'_i$. A category is ω-*complete* iff all ω-diagrams have colimits. Note that $\mathbf{Alg}_{\perp\vee E}$ and $(\mathbf{Alg}^{\mathsf{op}}_{\perp\vee})_E$ are ω-complete, and that if $\mathbf{C}_i$ are ω-complete then so is $\prod_i \mathbf{C}_i$.

A locally monotone functor $F : \mathbf{C} \to \mathbf{C}'$ between **DCPO**-enriched categories is ω-*continuous* iff $\mathbf{C}_E$ and $\mathbf{C}'_E$ are ω-complete, and the restriction $F : \mathbf{C}_E \to \mathbf{C}'_E$ preserves colimits of ω-diagrams.

For example, all of the locally continuous functors listed above are also continuous. In addition, if $F : \mathbf{C} \to \mathbf{C}'$, $G : \mathbf{C}' \to \mathbf{C}''$ and $H_i : \mathbf{C} \to \mathbf{C}_i$ are continuous, then so are F^{op},

$F;G$ and $\langle H_i \mid i \in I \rangle$.

A functor $F : \mathbf{C} \to \mathbf{C}$ has a *canonical* fixed point X iff $\mathsf{fold} : FX \to X$ is the initial F-algebra and $\mathsf{unfold} : X \to FX$ is the terminal F-coalgebra.

PROPOSITION 1. *Any continuous, locally continuous endofunctor on a* $\mathbf{DCPO}_\perp$*-enriched punctuated category has a canonical fixed point.*

PROOF. A generalization of the proofs in (Abramsky and Jung, 1994). □

For example, the domain equation for the lazy lambda-calculus is:

$$D \simeq (D \to D)_\perp$$

which can be found by taking the first element of the canonical fixed point of the functor:

$$\langle ((\_ \to \_);\_\perp)^{\mathsf{op}}, ((\_ \to \_);\_\perp) \rangle : \mathbf{Alg}_{\perp\vee}^{\mathsf{op}} \times \mathbf{Alg}_{\perp\vee} \to \mathbf{Alg}_{\perp\vee}^{\mathsf{op}} \times \mathbf{Alg}_{\perp\vee}$$

3 Sequential computation

This section shows how categorical structure can be used to model common programming structures. To do this, we construct a series of programming languages by gradually adding features, and showing that these features can be modelled categorically.

To show this formally, we shall borrow a notion from categorical algebra, and view programming languages as monads.

For example, given a set of values V ranged over by v, we can define $\mathsf{CL}\,V$ to be the *cat lists* over V, given by the grammar:

$$e ::= [v] \mid [] \mid e +\!\!+ e$$

up to the equivalence class given by:

$$e +\!\!+ [] = e = [] +\!\!+ e \qquad (e +\!\!+ f) +\!\!+ g = e +\!\!+ (f +\!\!+ g)$$

Then $\mathsf{CL}\,V$ is itself a set, so we can regard CL as a function from sets to sets. Moreover, given any function $F : V \to V'$ we can lift it to a function $\mathsf{CL}\,F : \mathsf{CL}\,V \to \mathsf{CL}\,V'$ as:

$$\mathsf{CL}\,Fv = Fv \qquad \mathsf{CL}\,F[] = [] \qquad \mathsf{CL}\,F(e +\!\!+ f) = (\mathsf{CL}\,Fe) +\!\!+ (\mathsf{CL}\,F\,f)$$

we can then verify that CL satisfies the criteria for being a functor:

$$\mathsf{CL}\,\mathsf{id} = \mathsf{id} \qquad \mathsf{CL}(F;G) = \mathsf{CL}\,F;\mathsf{CL}\,G$$

We also have an *injection* function $\eta : V \to \mathsf{CL}\,V$ and a *flattening* function $\mu : \mathsf{CL}(\mathsf{CL}\,V) \to \mathsf{CL}\,V$:

$$\eta v = [v] \qquad \mu[e] = e \qquad \mu[] = [] \qquad \mu(e +\!\!+ f) = \mu e +\!\!+ \mu f$$

These two functions satisfy the equations:

$$\eta;\mu = \mathsf{id} \qquad \mathsf{CL}\,\eta;\mu = \mathsf{id} \qquad \mathsf{CL}\,\mu;\mu = \mu;\mu$$

and so CL forms a *monad*.

We can then ask what a 'reasonable' model of CL would be. The criteria we consider here are:

- the model should contain a denotation for the singletons $[\_]$, and

- the model should be *denotational*, so if $[\![e]\!] = [\![f]\!]$ then $[\![C[e]]\!] = [\![C[f]]\!]$.

A model satisfies these criteria precisely when it is a CL-algebra, since these conditions correspond to respecting η and μ respectively. This means we can refine the informal question 'What are reasonable models of CL?' into the formal question 'What are the CL-algebras?'

Any CL-algebra must be a monoid, since we have a binary operation $\otimes$ with a unit I given by:

$$x \otimes y = [\![[x] +\!\!+ [y]]\!] \qquad I = [\![[]]\!]$$

Moreover, any monoid M is a CL-algebra, since we can define the denotational semantics of $\mathsf{CL}M$ as:

$$[\![[v]]\!] = v \qquad [\![[]]\!] = I \qquad [\![e +\!\!+ f]\!] = [\![e]\!] \otimes [\![f]\!]$$

We can express this one-to-one correspondence more precisely by showing that CL-**Alg** is *isomorphic* to **Mon**.

This example is one of the motivating uses of monads and algebras, and suggests a general technique for searching for models of programming languages:

- define a category for the basic values (in this case V is an object in **Set**),
- define a programming language parameterized by basic values (in this case CL is a functor on **Set**),
- show that the programming language forms a monad on the category (in this case using singletons and flattening), and
- find the category of algebras of the programming language (in this case **Mon**).

In this section we shall use the technique of finding categories of T-algebras to show the correspondence:

Programming construct	*Categorical equivalent*
Algebraic datatypes	Categories with finite products
Let-expressions	Strong monads
Functions	T-exponentials
If-then-else expressions	Computational coproducts of 1
Deconstructors	Morphisms in the Kleisli category

The results in this section are taken in part from Moggi's (Moggi, 1991) monadic meta-language, although the treatment of products, if-then-else statements, and deconstructors is rather different.

3.1 Algebraic datatypes

In this section, we shall present a simple language for algebraic datatypes, and show that its algebras (and hence its 'reasonable models') are precisely *categories with finite products*.

A (*many-sorted*) *signature* (ranged over by Σ) is a set of *sorts* (ranged over by A, B and C) and a set of *constructors* (ranged over by c) together with a *sorting* $c : A_1, \ldots, A_n \to A$.

For example, the signature $\mathsf{NatList}$ for lists of numbers has sorts bool, nat and list,

and constructors:

$$\begin{array}{rlrl} \mathsf{true} : & \to \mathsf{bool} & \mathsf{false} : & \to \mathsf{bool} \\ \mathsf{zero} : & \to \mathsf{nat} & \mathsf{succ} : & \mathsf{nat} \to \mathsf{nat} \\ \mathsf{nil} : & \to \mathsf{list} & \mathsf{cons} : & \mathsf{nat}, \mathsf{list} \to \mathsf{list} \end{array}$$

As another example, given sets $X_1, \ldots, X_n$, the signature $\Sigma_{X_1,\ldots,X_n}$ has sorts $X_1, \ldots, X_n$ and functions $f : X_{i_1} \times \cdots \times X_{i_n} \to X_i$ as constructors with sorting $f : X_{i_1}, \ldots, X_{i_n} \to X_i$.

A *signature morphism* $f : \Sigma \to \Sigma'$ consists of a mapping from the sorts of Σ to the sorts of Σ' and a mapping from the constructors of Σ to the constructors of Σ' such that whenever $c : A_1, \ldots, A_n \to A$ in Σ then $fc : fA_1, \ldots, fA_n \to fA$ in Σ'.

For example, there is a signature morphism $[\![\_]\!]$ from $\mathsf{NatList}$ to $\Sigma_{\{t,f\},\omega,\omega^*}$ which maps sorts as:

$$[\![\mathsf{bool}]\!] = \{t, f\} \qquad [\![\mathsf{nat}]\!] = \omega \qquad [\![\mathsf{list}]\!] = \omega^*$$

and maps constructors as:

$$\begin{array}{rlrl} [\![\mathsf{true}]\!] & = t & [\![\mathsf{false}]\!] & = f \\ [\![\mathsf{zero}]\!] & = 0 & [\![\mathsf{succ}]\!] & = \_ + 1 \\ [\![\mathsf{nil}]\!] & = \varepsilon & [\![\mathsf{cons}]\!] & = \_.\_ \end{array}$$

Let **Sig** be the category of signatures with signature morphisms.

Given a signature Σ, we can define the language $\mathsf{ST}\,\Sigma$ of *syntax trees* over Σ as:

$$e ::= * \mid c(e_1, \ldots, e_n) \mid (e, e) \mid v$$

where v ranges over *lvalues* given by the grammar:

$$v ::= x \mid v.\mathrm{L} \mid v.\mathrm{R}$$

where x ranges over a set of *variables*. These lvalues allow projections of pairs, as we shall see below.

The *closed* terms (those which contain no lvalues) are tuples of expressions built from Σ. For example, some open terms from $\mathsf{ST}\,\mathsf{NatList}$ are:

$$\mathsf{true}, \mathsf{false}$$

$$\mathsf{zero}, \mathsf{succ}\,\mathsf{zero}, \mathsf{succ}\,\mathsf{succ}\,\mathsf{zero}, \ldots$$

$$\mathsf{nil}, \mathsf{cons}(\mathsf{zero}, \mathsf{nil}), \mathsf{cons}(\mathsf{zero}, \mathsf{cons}(\mathsf{succ}\,\mathsf{zero}, \mathsf{nil})), \ldots$$

The *open* terms (those containing lvalues) contain free variables which may have terms *substituted* for them. In this section, the variables are just acting as place-holders, since there are no constructs for *binding* variables to values, but we shall add such a construct in the next section when we deal with let-expressions.

For example, some open terms from $\mathsf{ST}\,\mathsf{NatList}$ are:

$$x$$

$$\mathsf{succ}\,x, \mathsf{succ}\,\mathsf{succ}\,x, \ldots$$

$$\mathsf{cons}(x, y), \mathsf{cons}(x, \mathsf{cons}(\mathsf{succ}\,x, y)), \ldots$$

(we shall discuss the lvalues $v.\mathrm{L}$ and $v.\mathrm{R}$ below).

We can give $\mathsf{ST}\Sigma$ a static type system, with types:

$$\tau ::= I \mid [A] \mid \tau \otimes \tau$$

and type judgements of the form $\Gamma \vdash e : \tau$ given by rules:

$$\frac{}{\Gamma \vdash * : I} \qquad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash f : \tau}{\Gamma \vdash (e,f) : \sigma \otimes \tau}$$

$$\frac{\Gamma \vdash e_1 : [A_1] \quad \cdots \quad \Gamma \vdash e_n : [A_n]}{\Gamma \vdash c(e_1,\ldots,e_n) : [A]}[c : A_1,\ldots,A_n \to A]$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash y : \tau}{\Gamma, x : \sigma \vdash y : \tau}[x \neq y]$$

$$\frac{\Gamma \vdash v : (\sigma \otimes \tau)}{\Gamma \vdash v.\mathrm{L} : \sigma} \qquad \frac{\Gamma \vdash v : (\sigma \otimes \tau)}{\Gamma \vdash v.\mathrm{R} : \tau}$$

where Γ ranges over *contexts* of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$.

For example, we have the type judgements for closed terms:

$$\vdash \mathsf{true}, \mathsf{false} : [\mathsf{bool}]$$

$$\vdash \mathsf{zero}, \mathsf{succ}\,\mathsf{zero}, \mathsf{succ}\,\mathsf{succ}\,\mathsf{zero}, \ldots : [\mathsf{nat}]$$

$$\vdash \mathsf{nil}, \mathsf{cons}(\mathsf{zero}, \mathsf{nil}), \mathsf{cons}(\mathsf{zero}, \mathsf{cons}(\mathsf{succ}\,\mathsf{zero}, \mathsf{nil})), \ldots : [\mathsf{list}]$$

and for open terms:

$$x : [\mathsf{bool}] \vdash x : [\mathsf{bool}]$$

$$x : [\mathsf{nat}] \vdash \mathsf{succ}\,x, \mathsf{succ}\,\mathsf{succ}\,x, \ldots : [\mathsf{nat}]$$

$$x : [\mathsf{nat}], y : [\mathsf{list}] \vdash \mathsf{cons}(x,y), \mathsf{cons}(x, \mathsf{cons}(\mathsf{succ}\,x, y)), \ldots : [\mathsf{list}]$$

We can now explain the lvalues $v.\mathrm{L}$ and $v.\mathrm{R}$ as allowing the *projection* on pairs. For example:

$$z : [\mathsf{nat}] \otimes [\mathsf{list}] \vdash \mathsf{succ}(z.\mathrm{L}), \mathsf{succ}\,\mathsf{succ}(z.\mathrm{L}), \ldots : [\mathsf{nat}]$$

$$z : [\mathsf{nat}] \otimes [\mathsf{list}] \vdash \mathsf{cons}(z.\mathrm{L}, z.\mathrm{R}), \mathsf{cons}(z.\mathrm{L}, \mathsf{cons}(\mathsf{succ}\,z.\mathrm{L}, z.\mathrm{R})), \ldots : [\mathsf{list}]$$

Note that we are *not* allowing projections on arbitrary terms πe and $\pi' e$ (as would be more standard, for example in Moggi's (1991) monadic metalanguage) since this would not allow us to have the following useful properties:

- any term of type I is either an lvalue or $*$,
- any term of type $[A]$ is either an lvalue or of the form $c(e_1,\ldots,e_n)$, and
- any term of type $\sigma \otimes \tau$ is either an lvalue or of the form (e,f).

However, whenever $\Gamma \vdash e : \sigma \otimes \tau$, we can define $\Gamma \vdash \pi e : \sigma$ and $\Gamma \vdash \pi' e : \tau$ as syntactic sugar, since e must either be an lvalue v, in which case we define:

$$\pi v = v.\mathrm{L} \qquad \pi' v = v.\mathrm{R}$$

or e is a pair (f,g) in which case we define:

$$\pi(f,g) = f \qquad \pi'(f,g) = g$$

We are allowing multiple occurrences of one variable in a context, but only considering the right-most occurrence as significant. For example:

$$x : \sigma, x : \tau \vdash x : \tau$$
$$x : \sigma, x : \tau \not\vdash x : \sigma$$

$\mathsf{ST}\,\Sigma$ is itself a signature, with types as sorts and judgements of the form $(x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \tau)$ as constructors $\vec{\sigma} \to \tau$, viewed up to the congruence given by (when y is fresh):

$$\begin{aligned}
&(\Gamma \vdash x : I) \\
&\quad = (\Gamma \vdash * : I) && (1.\eta) \\
&(\Gamma \vdash (v.\mathrm{L}, v.\mathrm{R}) : \sigma \otimes \tau) \\
&\quad = (\Gamma \vdash v : \sigma \otimes \tau) && (\times.\eta) \\
&(\Gamma, x : \sigma, \Gamma' \vdash e : \tau) \\
&\quad = (\Gamma, y : \sigma, \Gamma' \vdash e[y/x] : \tau) && (\alpha)
\end{aligned}$$

Note that these equations only involve open terms, so closed terms are viewed up to syntactic identity. This is useful for the operational semantics given in Section 4, since we do not have to give the operational semantics up to an equivalence class on values.

We shall often elide the typing of terms where context makes it obvious. For any signature morphism $f : \Sigma \to \Sigma'$ we can define signature morphism $\mathsf{ST}\,f : \mathsf{ST}\,\Sigma \to \mathsf{ST}\,\Sigma'$ as:

$$\begin{aligned}
I &\mapsto I \\
[A] &\mapsto [fA] \\
\sigma \otimes \tau &\mapsto (\mathsf{ST}\,f\sigma) \otimes (\mathsf{ST}\,f\tau) \\
* &\mapsto * \\
c(e_1, \ldots, e_n) &\mapsto (fc)(\mathsf{ST}\,fe_1, \ldots, \mathsf{ST}\,fe_n) \\
(e, e') &\mapsto (\mathsf{ST}\,fe, \mathsf{ST}\,fe') \\
v &\mapsto v
\end{aligned}$$

It is routine to verify that $\mathsf{ST}\,f$ is a signature morphism and that $\mathsf{ST} : \mathbf{Sig} \to \mathbf{Sig}$ is a functor.

Whenever $\Gamma, \vec{x} : \vec{\sigma} \vdash e : \tau$ and $\Gamma \vdash \vec{f} : \vec{\sigma}$ we can define the *substitution* $\Gamma \vdash e[\vec{f}/\vec{x}] : \tau$ as usual, the only non-standard clause being:

$$v.\mathrm{L}[\vec{f}/\vec{x}] = \pi(v[f/x]) \quad v.\mathrm{R}[\vec{f}/\vec{x}] = \pi'(v[f/x])$$

For example:

$$\mathsf{cons}(z.\mathrm{L}, \mathsf{cons}(\mathsf{succ}\,z.\mathrm{L}, z.\mathrm{R}))[(\mathsf{zero}, \mathsf{nil})/z] = \mathsf{cons}(\mathsf{zero}, \mathsf{cons}(\mathsf{succ}\,\mathsf{zero}, \mathsf{nil}))$$

Following the outline given in the introduction to this section, we now show that $\mathsf{ST} : \mathbf{Sig} \to \mathbf{Sig}$ is a monad, using *injection* for η and *substitution* for μ.

We can define $\eta_\Sigma : \Sigma \to \mathsf{SL}\,\Sigma$ as:

$$\begin{aligned}
A &\mapsto [A] \\
(c : A_1, \ldots, A_n \to A) &\mapsto (x_1 : [A_1], \ldots, x_n : [A_n] \vdash c(x_1, \ldots, x_n) : [A])
\end{aligned}$$

and $\mu_\Sigma : \mathsf{SL}^2\,\Sigma \to \mathsf{SL}\,\Sigma$ as:

$$\begin{aligned} I &\mapsto I \\ [\tau] &\mapsto \tau \\ \sigma \otimes \tau &\mapsto \mu\sigma \otimes \mu\tau \\ * &\mapsto * \\ (\vec{x} : \vec{\sigma} \vdash e : \tau)(\vec{f}) &\mapsto e[\mu\vec{f}/\vec{x}] \\ (e, e') &\mapsto (\mu e, \mu e') \\ v &\mapsto v \end{aligned}$$

It is routine to verify that ST is a monad, and so the only remaining question is what are the ST-algebras? Proposition 2 tells us that these are equivalent to categories with finite products.

PROPOSITION 2. ST-**Alg** *is equivalent to* **CCat**.

PROOF. We need to provide two functors:

$$\mathsf{cat} : \mathsf{ST}\text{-}\mathbf{Alg} \to \mathbf{CCat} \qquad \mathsf{alg} : \mathbf{CCat} \to \mathsf{ST}\text{-}\mathbf{Alg}$$

and then show that we have natural isomorphisms:

$$\mathsf{cat}\,\mathsf{alg}\,\mathbf{C} \simeq \mathbf{C} \qquad \mathsf{alg}\,\mathsf{cat}\,\Sigma \simeq \Sigma$$

Given an ST-algebra Σ, let $\mathsf{cat}\,\Sigma$ be the category where objects are sorts and morphisms are unary constructors $c : A \to B$. This has categorical structure given by:

$$\begin{aligned} 1 &= [\![I]\!] \\ A \times B &= [\![[A] \otimes [B]]\!] \\ \mathsf{id} &= [\![x : [A] \vdash x : [A]]\!] \\ c; c' &= [\![x : [A] \vdash c'(c(x)) : [C]]\!] \\ ! &= [\![x : [A] \vdash * : I]\!] \\ \pi &= [\![x : [A] \otimes [B] \vdash x.\mathrm{L} : [A]]\!] \\ \pi' &= [\![x : [A] \otimes [B] \vdash x.\mathrm{R} : [B]]\!] \\ \langle c, c' \rangle &= [\![x : [A] \vdash (cx, c'x) : [B] \otimes [C]]\!] \end{aligned}$$

It is routine to verify that this makes $\mathsf{cat}\,\Sigma$ a category with finite products, and that any ST-algebra morphism $f : \Sigma \to \Sigma'$ lifts to a functor $\mathsf{cat}\,f : \mathsf{cat}\,\Sigma \to \mathsf{cat}\,\Sigma'$ which respects finite products.

Given a category with finite products $\mathbf{C}$, let $\mathsf{alg}\,\mathbf{C}$ be the signatures where sorts are objects and constructors are morphisms with the sorting $f : X_1, \ldots, X_n \to X$ whenever $f : X_1 \times \cdots \times X_n \to X$. This is an ST-algebra, since we can define the denotational semantics of $\mathsf{ST}(\mathsf{alg}\,\mathbf{C})$ by defining an object $[\![\tau]\!]$ in $\mathbf{C}$ as:

$$\begin{aligned} [\![I]\!] &= 1 \\ [\![\sigma \otimes \tau]\!] &= [\![\sigma]\!] \times [\![\tau]\!] \\ [\![[X]]\!] &= X \end{aligned}$$

an object $[\![\Gamma]\!]$ in $\mathbf{C}$ as:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

and a morphism $[\![\Gamma \vdash e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$ in $\mathbf{C}$ as (when $x \neq y$):

$$\begin{aligned}
[\![\Gamma \vdash * : I]\!] &= ! \\
[\![\Gamma \vdash (e, f) : \sigma \otimes \tau]\!] &= \langle [\![\Gamma \vdash e : \sigma]\!], [\![\Gamma \vdash f : \tau]\!] \rangle \\
[\![\Gamma \vdash f(e_1, \ldots, e_n) : [X]]\!] &= \langle [\![\Gamma \vdash e_1 : [X_1]]\!], \ldots, [\![\Gamma \vdash e_n : [X_n]]\!] \rangle ; f \\
[\![\Gamma, x : \sigma \vdash x : \sigma]\!] &= \pi' \\
[\![\Gamma, x : \sigma \vdash y : \tau]\!] &= \pi ; [\![\Gamma \vdash y : \tau]\!] \\
[\![\Gamma \vdash v.\mathrm{L} : \sigma]\!] &= [\![\Gamma \vdash v : \sigma \otimes \tau]\!] ; \pi \\
[\![\Gamma \vdash v.\mathrm{R} : \tau]\!] &= [\![\Gamma \vdash v : \sigma \otimes \tau]\!] ; \pi'
\end{aligned}$$

It is routine to verify that $\mathsf{alg}\,\mathbf{C}$ is a ST-algebra, and that any functor $F : \mathbf{C} \to \mathbf{C}'$ which respects finite products lifts to an ST-algebra morphism $\mathsf{alg}\,F : \mathsf{alg}\,\mathbf{C} \to \mathsf{alg}\,\mathbf{C}'$.

It is routine to verify that $\mathsf{alg}; \mathsf{cat} = \mathsf{id}$, but it is *not* the case that $\mathsf{cat}; \mathsf{alg} = \mathsf{id}$, since only unary constructors are preserved by cat. However, whenever two SL-algebras $(\Sigma, [\![\_]\!])$ and $(\Sigma', [\![\_]\!]')$ have the same sorts and unary constructors, we can show them to be isomorphic with the SL-algebra morphism:

$$\begin{aligned}
A &\mapsto A \\
(c : A_1, \ldots, A_n \to A) &\mapsto [\![x_1 : [A_1], \ldots, x_n : [A_n] \vdash c^*(\mathsf{tuple}(x_1, \ldots, x_n)) : [A]]\!]'
\end{aligned}$$

where $\mathsf{tuple} : A_1, \ldots, A_n \to A_1 \times \cdots \times A_n$ is:

$$\mathsf{tuple} = [\![y_1 : [A_1], \ldots, y_n : [A_n] \vdash (y_1, \ldots, y_n) : [A_1] \otimes \cdots \otimes [A_n]]\!]$$

and if $c : A_1, \ldots, A_n \to A$ then $c^* : A_1 \times \cdots \times A_n \to A$ is:

$$c^* = [\![x : [A_1 \times \cdots \times A_n] \vdash c(\pi_{n,1} x, \ldots, \pi_{n,n} x) : [A]]\!]$$

Moreover, we can show that this SL-algebra morphism is a natural isomorphism between $\mathsf{alg}(\mathsf{cat}\,\Sigma)$ and Σ, and so ST-**Alg** is equivalent to **CCat**. □

3.2 Monadic metalanguage

In the previous section, we saw that the appropriate categorical model for a simple language of data is categories with finite products. However, there was no mention of *computation* in that presentation, which we shall rectify in this section.

We shall follow Moggi (1991) in making two assumptions: that *let-expressions* are an appropriate primitive for computation, and that we should introduce a *type constructor* for computation.

To do this, we extend $\mathsf{ST}\,\Sigma$ to the *monadic metalanguage*, $\mathsf{MML}\,\Sigma$, by adding two new expression constructions:

$$e ::= \cdots \mid [e] \mid \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, e$$

These are:

- $[e]$ is a computation which immediately terminates with result e. For example, $[\mathsf{zero}]$

is a computation of an integer which immediately returns zero. This is similar to 'exit' in LOTOS (8807, 1989), and 'return' in CML.

- $\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f$ is a computation which evaluates e until it returns a value, which is then bound to x in f. For example, $\mathsf{let}\, x \Leftarrow [\mathsf{zero}] \,\mathsf{in}\, [\mathsf{succ}\, x]$ is the same as $[\mathsf{succ}\,\mathsf{zero}]$.

We also extend the type system by adding a new type constructor for computations:

$$\tau ::= \cdots \mid \mathsf{C}\,\tau$$

and statically typing $\mathsf{MML}\Sigma$ as:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] : \mathsf{C}\,\tau} \qquad \frac{\Gamma \vdash e : \mathsf{C}\,\sigma \quad \Gamma, x : \sigma \vdash f : \mathsf{C}\,\tau}{\Gamma \vdash \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f : \mathsf{C}\,\tau}$$

For example, we have:

$$\begin{aligned}
&\vdash [\mathsf{zero}] : \mathsf{C}\,\mathsf{nat} \\
&\vdash \mathsf{let}\, x \Leftarrow [\mathsf{zero}] \,\mathsf{in}\, [\mathsf{succ}\, x] : \mathsf{C}\,\mathsf{nat} \\
&\vdash [[\mathsf{zero}]] : \mathsf{C}\,\mathsf{C}\,\mathsf{nat} \\
&\vdash \mathsf{let}\, x \Leftarrow [[\mathsf{zero}]] \,\mathsf{in}\, x : \mathsf{C}\,\mathsf{nat}
\end{aligned}$$

Note in particular that we are allowing higher-order computations such as $[[\mathsf{zero}]]$, which immediately terminates with a *computation* $[\mathsf{zero}]$. This is similar to CML's event type constructor.

Then MML forms a monad in the same way as ST does, with the addition of Moggi's (1991) axioms (when x is not free in g):

$$\begin{aligned}
&(\Gamma \vdash \mathsf{let}\, y \Leftarrow f \,\mathsf{in}\, g : \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash \mathsf{let}\, x \Leftarrow f \,\mathsf{in}\, g[x/y] : \mathsf{C}\,\tau) && (\mathsf{C}\text{-}\alpha) \\
&(\Gamma \vdash \mathsf{let}\, x \Leftarrow [e] \,\mathsf{in}\, f : \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash f[e/x] : \mathsf{C}\,\tau) && (\mathsf{C}\text{-}\beta) \\
&(\Gamma \vdash \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, [x] : \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash e : \mathsf{C}\,\tau) && (\mathsf{C}\text{-}\eta) \\
&(\Gamma \vdash \mathsf{let}\, y \Leftarrow (\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f) \,\mathsf{in}\, g : \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, (\mathsf{let}\, y \Leftarrow f \,\mathsf{in}\, g) : \mathsf{C}\,\tau) && (\mathsf{C}\text{-ass})
\end{aligned}$$

The next proposition shows that the MML-algebras are precisely strong monads (hence the name 'monadic metalanguage'). This result is due largely to Moggi (1991).

PROPOSITION 3. MML-**Alg** *is equivalent to* **SMon**.

PROOF. For any MML-algebra Σ, let $\mathsf{cat}\,\Sigma$ have the monadic structure:

$$\begin{aligned}
TA &= [\![\mathsf{C}[A]]\!] \\
Tc &= [\![x : \mathsf{C}[A] \vdash \mathsf{let}\, y \Leftarrow x \,\mathsf{in}\, [cy] : \mathsf{C}[B]]\!] \\
\eta &= [\![x : [A] \vdash [x] : \mathsf{C}[A]]\!] \\
\mu &= [\![x : \mathsf{C}\,\mathsf{C}[A] \vdash \mathsf{let}\, y \Leftarrow x \,\mathsf{in}\, y : \mathsf{C}[A]]\!] \\
t &= [\![x : [A] \otimes \mathsf{C}[B] \vdash \mathsf{let}\, y \Leftarrow x.\mathrm{R} \,\mathsf{in}\, [(x.\mathrm{L}, y)] : \mathsf{C}([A] \otimes [B])]\!]
\end{aligned}$$

It is routine to verify that $\mathsf{cat}\,\Sigma$ is a strong monad. Given an MML-algebra morphism $f : \Sigma \to \Sigma'$ it is routine to verify that $\mathsf{cat}\,f : \mathsf{cat}\,\Sigma \to \mathsf{cat}\,\Sigma'$ preserves the strong monadic structure, and so is an **SMon** morphism.

For any strong monad $T : \mathbf{C} \to \mathbf{C}$, let $\mathsf{alg}\,\mathbf{C}$ be extended with semantics for MML given by:

$$\begin{aligned}
[\![\mathsf{C}\,\tau]\!] &= T[\![\tau]\!] \\
[\![\Gamma \vdash [e] : \mathsf{C}\,\tau]\!] &= [\![\Gamma \vdash e : \tau]\!];\eta \\
[\![\Gamma \vdash \mathsf{let}\,x \Leftarrow e\,\mathsf{in}\,f : \mathsf{C}\,\tau]\!] &= \langle \mathsf{id}, [\![\Gamma \vdash e : \mathsf{C}\,\sigma]\!]\rangle; t; T[\![\Gamma, x : \sigma \vdash f : \mathsf{C}\,\sigma]\!]; \mu
\end{aligned}$$

It is routine to verify that this is an MML-algebra, and that if $F : \mathbf{C} \to \mathbf{C}'$ preserves the monadic and cartesian structure, then $\mathsf{alg}\,F$ is an MML-algebra morphism.

It is routine to verify that alg and cat form an equivalence. □

3.3 Partial functions

The monadic metalanguage does not allow for any form of parameterized computation, such as procedures or functions. In this section, we extend $\mathsf{MML}\Sigma$ to a higher-order functional programming language, and show that the corresponding categorical structure is *computational cartesian closed categories* (*cccc*s). This development follows Moggi (1991), although the details are new.

The *functional* monadic metalanguage $\mathsf{MML}\lambda\Sigma$, extends $\mathsf{MML}\Sigma$ with expressions:

$$e ::= \cdots \mid \lambda x\,.\,e \mid ee$$

We also extend the type system by adding a new type constructor for functions:

$$\tau ::= \cdots \mid \tau \to \mathsf{C}\,\tau$$

and statically typing $\mathsf{MML}\lambda\Sigma$ as:

$$\frac{\Gamma, x : \sigma \vdash e : \mathsf{C}\,\tau}{\Gamma \vdash \lambda x\,.\,e : \sigma \to \mathsf{C}\,\tau} \qquad \frac{\Gamma \vdash e : \sigma \to \mathsf{C}\,\tau, f : \sigma}{\Gamma \vdash ef : \mathsf{C}\,\tau}$$

For example:

$$\begin{aligned}
&\vdash \lambda x\,.\,[\mathsf{succ}\,x] : \mathsf{nat} \to \mathsf{C}\,\mathsf{nat} \\
&\vdash (\lambda x\,.\,[\mathsf{succ}\,x])\mathsf{zero} : \mathsf{C}\,\mathsf{nat}
\end{aligned}$$

Note that we are only allowing functions to return computations, for example there is no type $\mathsf{nat} \to \mathsf{nat}$, only $\mathsf{nat} \to \mathsf{C}\,\mathsf{nat}$. This corresponds to our intuition that the only terms which involve computation are terms of computation type, and this would not be true if we allowed functions to return arbitrary type. This restriction also allows us to show that:

- any term of type I is either an lvalue or $*$,
- any term of type $[A]$ is either an lvalue or of the form $c(e_1, \ldots, e_n)$,
- any term of type $\sigma \otimes \tau$ is either an lvalue or of the form (e, f), and
- any term of type $\sigma \to \mathsf{C}\,\tau$ is either an lvalue or of the form $\lambda x\,.\,e$.

Note that we have *no* similar result about terms of type $\mathsf{C}\,\tau$.

Then $\mathsf{MML}\lambda$ forms a monad in the same way as MML does, with the addition of the

standard α, β and η axioms for functions (when y is not free in e):

$$\begin{aligned}
&(\Gamma \vdash \lambda x\,.\,e : \sigma \to \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash \lambda y\,.\,e[y/x] : \sigma \to \mathsf{C}\,\tau) && (\to\text{-}\alpha) \\
&(\Gamma \vdash (\lambda x\,.\,e)\,f : \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash e[f/x] : \mathsf{C}\,\tau) && (\to\text{-}\beta) \\
&(\Gamma \vdash \lambda y\,.\,(e y) : \sigma \to \mathsf{C}\,\tau) \\
&\quad = (\Gamma \vdash e : \sigma \to \mathsf{C}\,\tau) && (\to\text{-}\eta)
\end{aligned}$$

These axioms are those required to show that the models for typed λ-calculi are precisely *cartesian closed categories* (Lambek and Scott, 1986). Functional MML has a more restrictive type system, and so we have the corresponding restricted result that algebras for $\mathsf{MML}\lambda$ are ccccs.

PROPOSITION 4. $\mathsf{MML}\lambda$-**Alg** *is equivalent to* **CCCC**.

PROOF. For any $\mathsf{MML}\lambda$-algebra Σ, let $\mathsf{cat}\,\Sigma$ have T-exponentials give by:

$$TB^A = [\![[A] \to \mathsf{C}[B]]\!]$$

with curry given:

$$\begin{aligned}
\mathsf{curry}\,c &= [\![x : [A] \vdash \lambda y\,.\,[c(\mathsf{tuple}(x,y))] : [B] \to \mathsf{C}[TC]]\!]; \\
&\qquad [\![x : [B] \to \mathsf{C}\,\mathsf{C}[C] \vdash \lambda y\,.\,\mathsf{let}\,z \Leftarrow xy\,\mathsf{in}\,z : [B] \to \mathsf{C}[C]]\!] \\
\mathsf{curry}^{-1}\,c &= (c \times \mathsf{id}); \mathsf{apply} \\
\mathsf{apply} &= [\![x : ([B] \to \mathsf{C}[C]) \otimes [B] \vdash (x.\mathrm{L})(x.\mathrm{R}) : \mathsf{C}[C]]\!]
\end{aligned}$$

The tricky part of this proof is showing that curry is a natural bijection. This is difficult because the definition of curry involves an implicit type coercion, between the types $\mathsf{C}[TC]$ and $\mathsf{C}\,\mathsf{C}[C]$. These types have the same semantics (T^2C) but are syntactically different, but the bijection can be proved by equational reasoning using appropriate use of the fact that $[\![\_]\!]$ is an $\mathsf{MML}\lambda$-algebra.

It is routine to verify that $\mathsf{cat}\,\Sigma$ is a cccc, and that $\mathsf{cat}\,f : \mathsf{cat}\,\Sigma \to \mathsf{cat}\,\Sigma'$ is a cccc morphism.

For any cccc $T : \mathbf{C} \to \mathbf{C}$, let $\mathsf{alg}\,\mathbf{C}$ be extended with semantics for $\mathsf{MML}\lambda$ given by:

$$\begin{aligned}
[\![\sigma \to \mathsf{C}\,\tau]\!] &= T[\![\tau]\!]^{[\![\sigma]\!]} \\
[\![\Gamma \vdash \lambda x\,.\,e : \sigma \to \mathsf{C}\,\tau]\!] &= \mathsf{curry}[\![\Gamma, x : \sigma \vdash e : \mathsf{C}\,\tau]\!] \\
[\![\Gamma \vdash e\,f : \mathsf{C}\,\tau]\!] &= \langle [\![\Gamma \vdash e : \sigma \to \mathsf{C}\,\tau]\!], [\![\Gamma \vdash f : \sigma]\!] \rangle; \mathsf{ev}
\end{aligned}$$

It is routine to verify that $\mathsf{alg}\,\mathbf{C}$ is an $\mathsf{MML}\lambda$-algebra, and that $\mathsf{alg}\,F : \mathsf{alg}\,\mathbf{C} \to \mathsf{alg}\,\mathbf{C}'$ is an $\mathsf{MML}\lambda$-algebra morphism.

It is routine to verify that alg and cat form an equivalence. □

3.4 Control flow

One feature which is missing from functional MML is the ability for computation to depend on data. For example, it is impossible to implement an 'if-then-else' function of type $[\mathsf{bool}] \otimes [\mathsf{nat}] \otimes [\mathsf{nat}] \to \mathsf{C}[\mathsf{nat}]$ in $\mathsf{MML}\lambda\mathsf{list}$.

In this section we shall add a simple control flow operator, and show that it can be modelled by a restricted form of *coproducts*

A signature *with booleans* is a signature with a sort bool and constructors $\mathsf{true}, \mathsf{false} : () \to \mathsf{bool}$. Let **SigB** be the category of boolean signatures, together with morphisms which respect bool, true and false.

For any boolean signature Σ, the functional monadic metalanguage *with booleans* $\mathsf{MML}\lambda\mathsf{b}\Sigma$ extends $\mathsf{MML}\lambda\Sigma$ with expressions:

$$e ::= \cdots \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e$$

and with the type judgement:

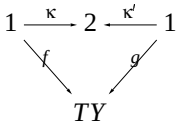
$$\frac{\Gamma \vdash e : [\mathsf{bool}], f : \mathsf{C}\tau, g : \mathsf{C}\tau}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : \mathsf{C}\tau}$$

Again, note that since 'if-then-else' statements require computation, they are restricted to terms of type $\mathsf{C}\tau$.

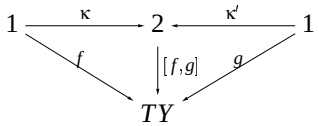
$\mathsf{MML}\lambda\mathsf{b}$ forms a monad on **SigB** in the same way as $\mathsf{MML}\lambda$ does, with the addition of axioms for 'if-then-else':

$$\begin{aligned}
&(\Gamma \vdash \mathsf{if}\, \mathsf{true}\, \mathsf{then}\, f\, \mathsf{else}\, g : \mathsf{C}\tau) \\
&\quad = (\Gamma \vdash f : \mathsf{C}\tau) && (\mathsf{if}\text{-}\beta) \\
&(\Gamma \vdash \mathsf{if}\, \mathsf{false}\, \mathsf{then}\, f\, \mathsf{else}\, g : \mathsf{C}\tau) \\
&\quad = (\Gamma \vdash g : \mathsf{C}\tau) && (\mathsf{if}\text{-}\beta') \\
&(\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, f : \mathsf{C}\tau) \\
&\quad = (\Gamma \vdash f : \mathsf{C}\tau) && (\mathsf{if}\text{-}\eta) \\
&(\Gamma \vdash \mathsf{if}\, x\, \mathsf{then}\, f[\mathsf{true}/x]\, \mathsf{else}\, g[\mathsf{false}/x] : \mathsf{C}\tau) \\
&\quad = (\Gamma \vdash \mathsf{if}\, x\, \mathsf{then}\, f\, \mathsf{else}\, g : \mathsf{C}\tau) && (\mathsf{if}\text{-}\eta')
\end{aligned}$$

A category with a strong monad $T : \mathbf{C} \to \mathbf{C}$ has *computational coproducts of* 1 iff there is a distinguished object 2, with maps $\kappa, \kappa' : 1 \to 2$ such that for any commuting diagram:

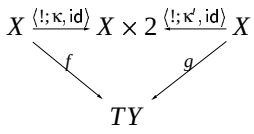


there is a unique mediating arrow $[f,g]$ such that:

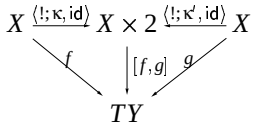


The category has *indexed* computational coproducts of 1 iff it has computational coprod-

ucts of 1 and for every commuting diagram:



there is a unique mediating arrow $[f,g]$ such that:



Note that any category with indexed computational coproducts of 1 must have coproducts of 1, since we can take X to be 1.

We shall show below that models of $\mathsf{MML}\lambda\mathsf{b}$ are precisely cccc's with computational coproducts of 1. First we shall show that in any cccc, any computational coproducts of 1 are indexed, and so we only need computational coproducts of 1 for a model of $\mathsf{MML}\lambda$ to be a model of $\mathsf{MML}\lambda\mathsf{b}$.

PROPOSITION 5. *Any cccc with computational coproducts of* 1 *has indexed computational coproducts of* 1.

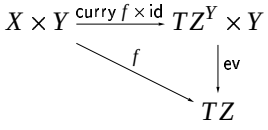
PROOF. For any $f,g : X \to TY$, let h be:

$$h = [\mathsf{curry}(\pi';f);\eta,\mathsf{curry}(\pi';g);\eta] : 2 \to T(TY^X)$$

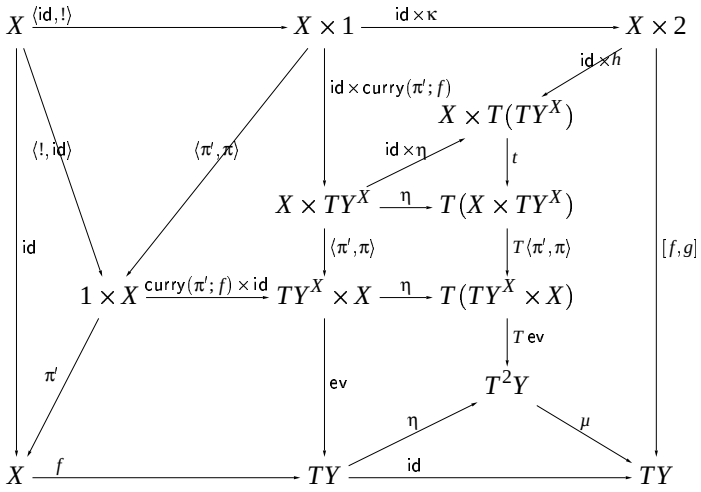
and let $[f,g]$ be:

$$X\times 2 \xrightarrow{\mathsf{id}\times h} X\times T(TY^X) \xrightarrow{t} T(X\times TY^X) \xrightarrow{T\langle\pi',\pi\rangle} T(TY^X\times X) \xrightarrow{T\,\mathsf{ev}} T^2Y \xrightarrow{\mu} TY$$

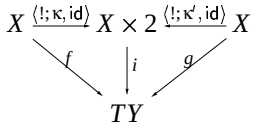
Then we use the fact that, for any $f : X\times Y \to TZ$, the diagram:



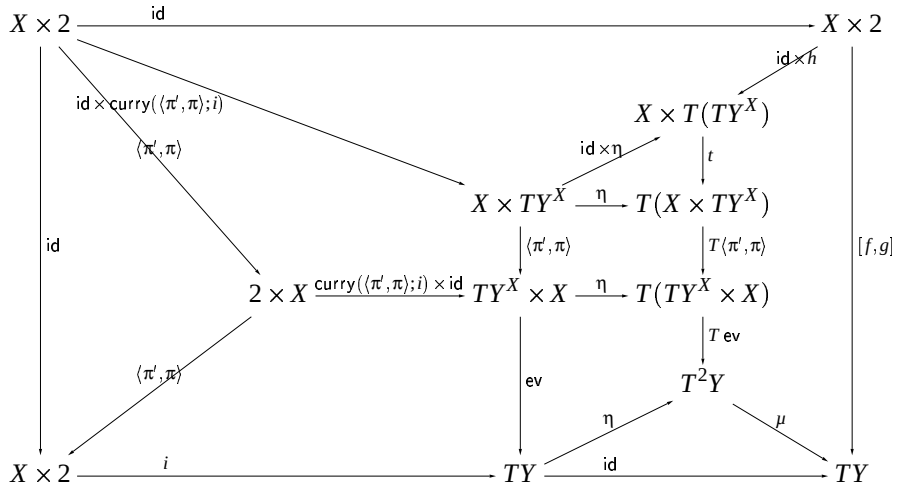
commutes to show that:



and similarly for κ', and so the indexed coproduct diagram commutes. For any other i which makes the indexed coproduct diagram commute:



we can show that $h = \mathsf{curry}(\langle \pi', \pi \rangle; i); \eta$, and thus:



Thus the category has indexed partial coproducts of 1. □

Let **CCCCB** be the subcategory of **CCCC** of categories with partial coproducts of 1, together with functors which respect 2, κ and κ'.

PROPOSITION 6. MMLλb-**Alg** *is equivalent to* **CCCCB**.

PROOF. For any MMLλb-algebra Σ, let $\mathsf{cat}\,\Sigma$ have the structure:

$$\begin{aligned} 2 &= \mathsf{bool} \\ \kappa &= [\![x : I \vdash \mathsf{true} : [\mathsf{bool}]]\!] \\ \kappa' &= [\![x : I \vdash \mathsf{false} : [\mathsf{bool}]]\!] \\ [c, c'] &= [\![x : [A] \otimes [\mathsf{bool}] \vdash \mathsf{if}\, x.\mathrm{R}\, \mathsf{then}\, [c(x.\mathrm{L})]\, \mathsf{else}\, [c'(x.\mathrm{L})] : \mathsf{C}[TB]]\!]; \mu \end{aligned}$$

It is routine to verify that these satisfy the defining conditions of an indexed partial coproduct of 1, and that $\mathsf{cat}\, f : \mathsf{cat}\,\Sigma \to \mathsf{cat}\,\Sigma'$ is an **CCCCB** morphism.

For any category **C** with indexed partial coproducts of 1, let $\mathsf{alg}\,\mathbf{C}$ be extended with semantics for MMLλb given by:

$$\begin{aligned} [\![\mathsf{bool}]\!] &= 2 \\ [\![\Gamma \vdash \mathsf{true} : [\mathsf{bool}]]\!] &= !; \kappa \\ [\![\Gamma \vdash \mathsf{false} : [\mathsf{bool}]]\!] &= !; \kappa' \\ [\![\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : \mathsf{C}\tau]\!] &= \langle \mathsf{id}, [\![\Gamma \vdash e : [\mathsf{bool}]]\!] \rangle; [[\![\Gamma \vdash f : \mathsf{C}\tau]\!], [\![\Gamma \vdash g : \mathsf{C}\tau]\!]] \end{aligned}$$

It is routine to verify that $\mathsf{alg}\,\mathbf{C}$ is an MMLλb-algebra, and that $\mathsf{alg}\, F : \mathsf{alg}\,\mathbf{C} \to \mathsf{alg}\,\mathbf{C}'$ is an MMLλb-algebra morphism.

It is routine to verify that alg and cat form an equivalence. □

3.5 Deconstructors

Although $\mathsf{MML}\lambda\mathsf{b}\,\Sigma$ allows computation to be affected by data, this is only allowed for expressions of type $[\mathsf{bool}]$. For example, although we can implement an equality test function of type $[\mathsf{bool}] \otimes [\mathsf{bool}] \to \mathsf{C}[\mathsf{bool}]$, we cannot implement an equality-test function of type $[\mathsf{nat}] \otimes [\mathsf{nat}] \to \mathsf{C}[\mathsf{bool}]$ in $\mathsf{MML}\lambda\mathsf{b}\,\mathsf{NatList}$.

This can be rectified by allowing signatures to have *deconstructors* as well as constructors.

A signature with booleans and *deconstructors* is a signature with booleans together with a set of deconstructors, ranged over by d, with sorting $d : A_1, \ldots, A_n \to B$.

For example, we can extend $\mathsf{NatList}$ to being a signature with booleans and deconstructors by adding deconstructors:

$$\begin{array}{ccc} \mathsf{eq} : \mathsf{nat}, \mathsf{nat} \to \mathsf{bool} & & \mathsf{pred} : \mathsf{nat} \to \mathsf{nat} \\ \mathsf{isnil} : \mathsf{list} \to \mathsf{bool} & \mathsf{hd} : \mathsf{list} \to \mathsf{nat} & \mathsf{tl} : \mathsf{list} \to \mathsf{list} \end{array}$$

In this section, we will not consider the semantics of deconstructors, and we shall leave that to Section 4.2.

A morphism $f : \Sigma \to \Sigma'$ between signatures with booleans and deconstructors is a mapping between sorts, constructors and deconstructors which respects the boolean structure and sorting. Let **SigBD** be the category of signatures with booleans and deconstructors.

For any signature with booleans deconstructors Σ, the functional monadic metalanguage with booleans deconstructors $\mathsf{MML}\lambda\mathsf{bd}\,\Sigma$ extends $\mathsf{MML}\lambda\mathsf{b}\,\Sigma$ with expressions:

$$e ::= \cdots \mid d\vec{e}$$

and with type judgements:

$$\frac{\Gamma \vdash \vec{e} : [\vec{A}]}{\Gamma \vdash d\vec{e} : \mathsf{C}[B]}[d : \vec{A} \to B]$$

For example, we can define an equality test for booleans as:

$$\begin{array}{l} \mathsf{eq} \;:\; [\mathsf{bool}] \otimes [\mathsf{bool}] \to \mathsf{C}[\mathsf{bool}] \\ \mathsf{eq} = \lambda x\,.\,\mathsf{if}\, x.\mathrm{L} \\ \qquad\quad \mathsf{then}[x.\mathrm{R}] \\ \qquad\quad \mathsf{else}\,\mathsf{not}(x.\mathrm{R}) \end{array}$$

where not is the negation function:

$$\begin{array}{l} \mathsf{not} \;:\; [\mathsf{bool}] \to \mathsf{C}[\mathsf{bool}] \\ \mathsf{not} = \lambda x\,.\,\mathsf{if}\, x \\ \qquad\quad \mathsf{then}[\mathsf{false}] \\ \qquad\quad \mathsf{else}[\mathsf{true}] \end{array}$$

$\mathsf{MML}\lambda\mathsf{bd}\,\Sigma$ is itself a signature with booleans and deconstructors:

$$(\vec{x} : \vec{\sigma} \vdash e : \mathsf{C}\,\tau) : \vec{\sigma} \to \tau$$

and we can show that $\mathsf{MML}\lambda\mathsf{bd}$ is a monad on **SigBD** in the same way as $\mathsf{MML}\lambda\mathsf{b}$.

The next proposition shows that the $\mathsf{MML}\lambda\mathsf{bd}$-algebras are precisely the same as the $\mathsf{MML}\lambda\mathsf{b}$-algebras. This may seem rather surprising, since we have added extra structure

to MMLλb, and we might expect to see this structure occurring in the categorical models for MMLλbd. However, it turns out that **CCCCB** already has enough structure, since the deconstructors can be modelled as morphisms in the *Kleisli category* of a strong monad, that is where a constructor $c : A \to B$ is a morphism of type $A \to B$, a deconstructor $d : A \to B$ is a morphism of type $A \to TB$. The deconstructors in an MMLλbd-algebra form a category with composition and identity given by:

$$\mathsf{id} = [\![x : [A] \vdash [x] : \mathsf{C}[A]]\!]$$
$$d; d' = [\![x : [A] \vdash \mathsf{let}\, y \Leftarrow dx \,\mathsf{in}\, d'y : \mathsf{C}[C]]\!]$$

PROPOSITION 7. MMLλbd-**Alg** *is equivalent to* **CCCCB**.

PROOF. It suffices to show that MMLλbd-**Alg** is equivalent to MMLλb-**Alg**. To do this, we show that MMLλbd-algebras are uniquely determined (up to isomorphism) by their constructors.

In any MMLλbd-algebra Σ, define the deconstructor:

$$(\partial : TA \to A) = [\![x : \mathsf{C}[A] \vdash x : \mathsf{C}[A]]\!]$$

and given a constructor $c : \vec{A} \to TB$, define the deconstructor:

$$(c_* : \vec{A} \to B) = [\![\vec{x} : [\vec{A}] \vdash \partial(c\vec{x}) : \mathsf{C}[B]]\!]$$

Given a deconstructor $d : \vec{A} \to B$, define the constructor:

$$(d^* : \vec{A} \to TB) = [\![\vec{x} : [\vec{A}] \vdash d\vec{x} : \mathsf{C}[B]]\!]$$

Then $(c_*)^* = c$ and $(d^*)_* = d$. From this it is routine to show that if two MMLλbd-algebras are isomorphic in **SigB** then they are isomorphic in **SigBD**, since we can extend the isomorphism i to deconstructors as:

$$i(d) = (i(d^*))_*$$

Thus MMLλbd-**Alg** is equivalent to MMLλb-**Alg** and hence to **CCCCB**. □

4 Nondeterminism

The work in Section 3 shows the precise correspondence between categorical models and programming languages, and in particular between strong monads and computation.

In this section we look at a particular strong monad, the *lower powerdomain* monad $\mathcal{P}$ on *algebraic dcpo's*, and show that it provides a *fully abstract* model for nondeterministic computation. That is, we show that the preorder on terms given by the denotational semantics is exactly the same as the *may-testing* pre-order defined operationally.

Powerdomains have long been used as models for concurrency, notably by Plotkin (1981, for example). Powerdomains over algebraic dcpo's form a cccc with computational coproducts of 1, which means that for free we have a model for functional MML with booleans and deconstructors. Hennessy and Plotkin (1979) and Mislove and Oles (1992) have shown techniques for proving full abstraction of powerdomain semantics. In this paper we show another technique, based on Abramsky's (1991) *domain theory in logical form*.

Domain theory in logical form uses a program logic as a stepping stone between the operational and denotational views of programs, and has been used by the author (1994) to show full abstraction for a concurrent call-by-need λ-calculus, and Hennessy (1992) to show full abstraction for a higher-order concurrent language based on Thomsen's (1989) CHOCS.

One corollary of the full abstraction result is that whenever two terms are denotatinally different, we can provide the reason why they are different. This reason can either be given as a context in which one term deadlocks where the other may terminate, or it can be given as a proposition, similar to the distinguishing formulae produced by verification tools such as TAV (Larsen *et al.*, 1989).

4.1 Syntax

The language we shall consider in this section is an extension of functional MML with booleans and deconstructors. We extend it with a syntax for recursion, and for nondeterminism.

Given a signature Σ with deconstructors and booleans, the *nondeterministic* monadic metalanguage $\mathsf{NMML}\Sigma$ extends $\mathsf{MML}\lambda\mathsf{bd}\Sigma$ with expressions:

$$e ::= \cdots \mid \delta \mid e \,\Box\, e \mid \mathsf{fix}(x = e)$$

and type judgements:

$$\frac{}{\Gamma \vdash \delta : \mathsf{C}\tau} \qquad \frac{\Gamma \vdash e : \mathsf{C}\tau, f : \mathsf{C}\tau}{\Gamma \vdash e \,\Box\, f : \mathsf{C}\tau} \qquad \frac{\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau}{\Gamma \vdash \mathsf{fix}(x = e) : \mathsf{C}\tau}$$

Note that we have only defined recursion on computations rather than on functions. However, we can define recursive functions as syntactic sugar:

$$\mathsf{fix}(x = \lambda y\,.\,e) = \lfloor \mathsf{fix}(z = (\lambda x\,.\,[\lambda y\,.\,e])\lfloor z \rfloor) \rfloor$$

where:

$$\lfloor e \rfloor = \lambda y\,.\,\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, xy$$

These have typing:

$$\frac{\Gamma \vdash e : \mathsf{C}(\sigma \to \mathsf{C}\tau)}{\Gamma \vdash \lfloor e \rfloor : \sigma \to \mathsf{C}\tau} \qquad \frac{\Gamma, x : \sigma \to \mathsf{C}\tau, y : \sigma \vdash e : \mathsf{C}\tau}{\Gamma \vdash \mathsf{fix}(x = \lambda y\,.\,e) : \sigma \to \mathsf{C}\tau}$$

We shall see below that this has the expected operational semantics:

$$\mathsf{fix}(x = \lambda y\,.\,e)\, f \Longrightarrow e[\mathsf{fix}(x = \lambda y\,.\,e)/x][f/y]$$

We will write $\lambda(\vec{x})\,.\,e$ as syntactic sugar, for example:

$$\lambda(x, y)\,.\,e = \lambda z\,.\,e[z.\mathrm{L}/x, z.\mathrm{R}/y]$$

and we will write $e(\vec{x}) \stackrel{\mathsf{def}}{=} C[e]$ as short for defining e to be $\mathsf{fix}(y = \lambda(\vec{x})\,.\,C[y])$.

For example, a recursive function to add an element to the end of a list is:

$$\mathsf{snoc} \;:\; [\mathsf{list}] \otimes [\mathsf{nat}] \to \mathsf{C}[\mathsf{list}]$$

$$\begin{array}{rl}\mathsf{snoc}(xs,x) \stackrel{\mathsf{def}}{=} & \mathsf{let}\, x \Leftarrow \mathsf{isnil}(xs)\, \mathsf{in} \\ & \mathsf{if}\, x \\ & \mathsf{then}[\mathsf{cons}(x,\mathsf{nil})] \\ & \mathsf{else}\, \mathsf{let}\, y \Leftarrow \mathsf{hd}(xs)\, \mathsf{in} \\ & \quad \mathsf{let}\, y' \Leftarrow \mathsf{tl}(xs)\, \mathsf{in} \\ & \quad \mathsf{let}\, z \Leftarrow \mathsf{snoc}(y',x)\, \mathsf{in} \\ & \quad [\mathsf{cons}(y,z)]\end{array}$$

This can be made tail-recursive by defining a function to reverse a list:

$$\begin{array}{rl}\mathsf{rev}' & : [\mathsf{list}] \otimes [\mathsf{list}] \to \mathsf{C}[\mathsf{list}] \\ \mathsf{rev}'(xs,ys) \stackrel{\mathsf{def}}{=} & \mathsf{let}\, x \Leftarrow \mathsf{isnil}(xs)\, \mathsf{in} \\ & \mathsf{if}\, x \\ & \mathsf{then}[ys] \\ & \mathsf{else}\, \mathsf{let}\, y \Leftarrow \mathsf{hd}(xs)\, \mathsf{in} \\ & \quad \mathsf{let}\, y' \Leftarrow \mathsf{tl}(xs)\, \mathsf{in} \\ & \quad \mathsf{rev}'(y', \mathsf{cons}(y,ys)) \\ \mathsf{rev} & : [\mathsf{list}] \to \mathsf{C}[\mathsf{list}] \\ \mathsf{rev}(xs) \stackrel{\mathsf{def}}{=} & \mathsf{rev}'(xs, \mathsf{nil})\end{array}$$

and then defining:

$$\begin{array}{rl}\mathsf{snoc}' & : [\mathsf{list}] \otimes [\mathsf{nat}] \to \mathsf{C}[\mathsf{list}] \\ \mathsf{snoc}'(xs,x) \stackrel{\mathsf{def}}{=} & \mathsf{let}\, ys \Leftarrow \mathsf{rev}(xs)\, \mathsf{in}\, \mathsf{rev}(\mathsf{cons}(x,ys))\end{array}$$

Note that the difference between tail-recursive and non-tail-recursive functions is made very apparent by the explicit use of let to control flow of execution.

The choice operator $e \,\Box\, f$ is based on CSP's (Hoare, 1985) external choice and so the choice is not made between e and f until they return a result. (This choice operator is used because it gives an appropriate semantics in the concurrent language (Jeffrey, 1995).) This means that even up to weak bisimulation (defined in Section 4.3) we have the equivalences:

$$e \,\Box\, \delta = e = \delta \,\Box\, e \qquad (e \,\Box\, f) \,\Box\, g = e \,\Box\, (f \,\Box\, g)$$

These equivalences allow us to model nondeterminism with a powerdomain model, since we can view δ as the empty set of results, $[e]$ as a singleton, and $\Box$ as union.

We have also only provided CSP (Hoare, 1985) external choice, and not internal choice. However, this can be defined:

$$e \sqcap f = \mathsf{let}\, x \Leftarrow [e] \,\Box\, [f]\, \mathsf{in}\, x$$

This has typing:

$$\frac{\Gamma \vdash e : \mathsf{C}\tau \quad \Gamma \vdash f : \mathsf{C}\tau}{\Gamma \vdash e \sqcap f : \mathsf{C}\tau}$$

The operational semantics for internal choice is given below as:

$$e \sqcap f \Longrightarrow e \qquad e \sqcap f \Longrightarrow f$$

As we shall see, internal and external choice cannot be distinguished by may-testing, but they can be distinguished by bisimulation.

4.2 Operational semantics

In this section we define the operational semantics of $\mathsf{NMML}\Sigma$.

In order to give an operational semantics for $\mathsf{NMML}\Sigma$, we need an operational semantics for the deconstructors of Σ. This is given as a *higher-order unlabelled value production system*, that is:

- an *internal transition* relation $e \xrightarrow{\imath} e'$, and
- a *termination* relation $e \xrightarrow{\surd} e'$

such that:

- if $e \xrightarrow{\imath} e'$ then $\vdash e : \mathsf{C}\tau$ and $\vdash e' : \mathsf{C}\tau$ for some τ,
- if $e \xrightarrow{\surd} e'$ then $\vdash e : \mathsf{C}\tau$ and $\vdash e' : \tau$ for some τ,
- $\xrightarrow{\surd}$ is deterministic, and
- if $e \xrightarrow{\surd}$ then $e \not\xrightarrow{\imath}$.

For example, the operational semantics for $\mathsf{NatList}$ is (when $e \neq f$):

$$\mathsf{eq}(e,e) \xrightarrow{\surd} \mathsf{true} \qquad \mathsf{eq}(e,f) \xrightarrow{\surd} \mathsf{false}$$
$$\mathsf{isnil}(\mathsf{nil}) \xrightarrow{\surd} \mathsf{true} \qquad \mathsf{isnil}(\mathsf{cons}(e,f)) \xrightarrow{\surd} \mathsf{false}$$
$$\mathsf{hd}(\mathsf{cons}(e,f)) \xrightarrow{\surd} e \qquad \mathsf{tl}(\mathsf{cons}(e,f)) \xrightarrow{\surd} f$$
$$\mathsf{pred}(\mathsf{succ}\, e) \xrightarrow{\surd} e$$

Note that we have not given any reductions for $\mathsf{pred}\,\mathsf{zero}$, $\mathsf{hd}\,\mathsf{nil}$ or $\mathsf{tl}\,\mathsf{nil}$, and so they deadlock.

A sort A in a signature with deconstructors and booleans is an *equality sort* iff there is a deconstructor $\mathsf{eq} : A, A \to \mathsf{bool}$ with operational semantics (when $e \neq f$):

$$\mathsf{eq}(e,e) \xrightarrow{\surd} \mathsf{true} \qquad \mathsf{eq}(e,f) \xrightarrow{\surd} \mathsf{false}$$

For example, nat is an equality sort in $\mathsf{NatList}$, but list is not.

Given an operational semantics for terms of the form $d\vec{e}$, we can extend it to an operational semantics for closed terms of $\mathsf{NMML}\Sigma$ with:

$$\frac{}{[e] \xrightarrow{\surd} e} \qquad \frac{e \xrightarrow{\imath} e'}{\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f \xrightarrow{\imath} \mathsf{let}\, x \Leftarrow e' \,\mathsf{in}\, f} \qquad \frac{e \xrightarrow{\surd} g}{\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f \xrightarrow{\imath} f[g/x]}$$

$$\frac{}{\mathsf{if}\,\mathsf{true}\,\mathsf{then}\, f\,\mathsf{else}\, g \xrightarrow{\imath} f} \qquad \frac{}{\mathsf{if}\,\mathsf{false}\,\mathsf{then}\, f\,\mathsf{else}\, g \xrightarrow{\imath} g}$$

$$\frac{}{(\lambda x\,.\,e)\, f \xrightarrow{\imath} e[f/x]} \qquad \frac{}{\mathsf{fix}(x = e) \xrightarrow{\imath} e[\mathsf{fix}(x = e)/x]}$$

$$\frac{e \xrightarrow{\imath} e'}{e \,\Box\, f \xrightarrow{\imath} e' \,\Box\, f} \qquad \frac{f \xrightarrow{\imath} f'}{e \,\Box\, f \xrightarrow{\imath} e \,\Box\, f'} \qquad \frac{e \xrightarrow{\surd} e'}{e \,\Box\, f \xrightarrow{\imath} [e']} \qquad \frac{f \xrightarrow{\surd} f'}{e \,\Box\, f \xrightarrow{\imath} [f']}$$

PROPOSITION 8. *The operational semantics for* $\mathsf{NMML}\Sigma$ *is a higher-order unlabelled value production system.*

PROOF. Show by induction on the proof of $e \xrightarrow{\mu} e'$ that if $e \xrightarrow{\mu} e'$ then the conditions for a higher-order unlabelled vps are satisfied. □

Let $e \Longrightarrow e'$ iff $e \xrightarrow{\imath}{}^{*} e'$ and $e \overset{\surd}{\Longrightarrow} f$ iff $e \Longrightarrow \xrightarrow{\surd} f$.

For example, the operational semantics of $\mathsf{fix}(x = \lambda y.\, e)$ is:

$$
\begin{array}{l}
\mathsf{fix}(x = \lambda y.\, e)\, f \\
\quad \xrightarrow{\imath} \mathsf{let}\, w \Leftarrow \mathsf{fix}(z = (\lambda x.\, [\lambda y.\, e])\lfloor z \rfloor)\, \mathsf{in}\, w f \\
\quad \xrightarrow{\imath} \mathsf{let}\, w \Leftarrow (\lambda x.\, [\lambda y.\, e])(\mathsf{fix}(x = \lambda y.\, e))\, \mathsf{in}\, w f \\
\quad \xrightarrow{\imath} \mathsf{let}\, w \Leftarrow [\lambda y.\, e[\mathsf{fix}(x = \lambda y.\, e)/x]]\, \mathsf{in}\, w f \\
\quad \xrightarrow{\imath} (\lambda y.\, e[\mathsf{fix}(x = \lambda y.\, e)/x])\, f \\
\quad \xrightarrow{\imath} e[\mathsf{fix}(x = \lambda y.\, e)/x][f/y]
\end{array}
$$

Thus the operational semantics of $\mathsf{snoc}(\mathsf{nil}, g)$ is:

$$
\begin{array}{l}
\mathsf{snoc}(\mathsf{nil}, g) \\
\quad \Longrightarrow \mathsf{let}\, x \Leftarrow \mathsf{isnil}\, \mathsf{nil}\, \mathsf{in} \\
\qquad \mathsf{if}\, x \\
\qquad \mathsf{then}\, [\mathsf{cons}(g, \mathsf{nil})] \\
\qquad \mathsf{else}\, \mathsf{let}\, y \Leftarrow \mathsf{hd}(\mathsf{nil})\, \mathsf{in} \\
\qquad\quad \mathsf{let}\, y' \Leftarrow \mathsf{tl}(\mathsf{nil})\, \mathsf{in} \\
\qquad\quad \mathsf{let}\, z \Leftarrow \mathsf{snoc}(y', g)\, \mathsf{in} \\
\qquad\quad [\mathsf{cons}(y, z)] \\
\quad \xrightarrow{\imath} \mathsf{if}\, \mathsf{true} \\
\qquad \mathsf{then}\, [\mathsf{cons}(g, \mathsf{nil})] \\
\qquad \mathsf{else}\, \mathsf{let}\, y \Leftarrow \mathsf{hd}(\mathsf{nil})\, \mathsf{in} \\
\qquad\quad \mathsf{let}\, y' \Leftarrow \mathsf{tl}(\mathsf{nil})\, \mathsf{in} \\
\qquad\quad \mathsf{let}\, z \Leftarrow \mathsf{snoc}(y', g)\, \mathsf{in} \\
\qquad\quad [\mathsf{cons}(y, z)] \\
\quad \xrightarrow{\imath} [\mathsf{cons}(g, \mathsf{nil})] \\
\quad \xrightarrow{\surd} \mathsf{cons}(g, \mathsf{nil})
\end{array}
$$

and if $\mathsf{snoc}(f, g) \overset{\surd}{\Longrightarrow} h$ then the operational semantics of $\mathsf{snoc}(\mathsf{cons}(e, f), g)$ is:

$$
\begin{array}{l}
\mathsf{snoc}(\mathsf{cons}(e, f), g) \\
\quad \Longrightarrow \mathsf{let}\, x \Leftarrow \mathsf{isnil}(\mathsf{cons}(e, f), g)\, \mathsf{in} \\
\qquad \mathsf{if}\, x \\
\qquad \mathsf{then}\, [\mathsf{cons}(g, \mathsf{cons}(e, f))] \\
\qquad \mathsf{else}\, \mathsf{let}\, y \Leftarrow \mathsf{hd}(\mathsf{cons}(e, f))\, \mathsf{in} \\
\qquad\quad \mathsf{let}\, y' \Leftarrow \mathsf{tl}(\mathsf{cons}(e, f))\, \mathsf{in} \\
\qquad\quad \mathsf{let}\, z \Leftarrow \mathsf{snoc}(y', g)\, \mathsf{in} \\
\qquad\quad [\mathsf{cons}(y, z)]
\end{array}
$$

$$
\begin{array}{ll}
\xrightarrow{\iota} & \mathsf{if}\,\mathsf{false} \\
& \mathsf{then}[\mathsf{cons}(g,\mathsf{cons}(e,f))] \\
& \mathsf{else}\,\mathsf{let}\,y \Leftarrow \mathsf{hd}(\mathsf{cons}(e,f))\mathsf{in} \\
& \quad \mathsf{let}\,y' \Leftarrow \mathsf{tl}(\mathsf{cons}(e,f))\mathsf{in} \\
& \quad \mathsf{let}\,z \Leftarrow \mathsf{snoc}(y',g)\mathsf{in} \\
& \quad [\mathsf{cons}(y,z)] \\
\xrightarrow{\iota} & \mathsf{let}\,y \Leftarrow \mathsf{hd}(\mathsf{cons}(e,f))\mathsf{in} \\
& \mathsf{let}\,y' \Leftarrow \mathsf{tl}(\mathsf{cons}(e,f))\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{snoc}(y',g)\mathsf{in} \\
& [\mathsf{cons}(y,z)] \\
\xrightarrow{\iota} & \mathsf{let}\,y' \Leftarrow \mathsf{tl}(\mathsf{cons}(e,f))\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{snoc}(y',g)\mathsf{in} \\
& [\mathsf{cons}(e,z)] \\
\xrightarrow{\iota} & \mathsf{let}\,z \Leftarrow \mathsf{snoc}(f,g)\mathsf{in} \\
& [\mathsf{cons}(e,z)] \\
\Longrightarrow & [\mathsf{cons}(e,h)] \\
\xrightarrow{\surd} & \mathsf{cons}(e,h)
\end{array}
$$

Thus we can show by induction that $\mathsf{snoc}(e,f)$ returns the list e with f appended to the end.

Note that this operational semantics explicitly represents many intermediate states of a computation which would normally be elided in an operational semantics. This is the price of making the flow of computation explicit using let-expressions. The advantage of doing so is a simpler operational semantics, and one which is 'closer to the metal' of an abstract machine.

There are a large number of possible operational equivalences and preorders which can be used to relate nondeterministic terms. In the rest of this section we shall concentrate on only one of them—*may testing*.

May-testing has been investigated by Hennessy for both first-order (1988) and higher-order (1992) untyped processes. It was first suggested as a model for the untyped λ-calculus by Morris (1968).

The assumption behind may-testing is that we are only interested in the observable external behaviour of terms, and moreover the only behaviour we are interested in is whether a process *may* terminate. For a full discussion of may- and must-testing for concurrent systems, see Hennessy's (1988) textbook.

For any $\Gamma \vdash e, f : \tau$, define the *may-testing preorder* as $\Gamma \models e \sqsubseteq_O f : \tau$ iff $C[e] \stackrel{\surd}{\Longrightarrow} *$ implies $C[f] \stackrel{\surd}{\Longrightarrow}$ for any closing context C of type $\mathsf{C}\,I$.

For example, here are some terms which are *not* may-testing equivalent, together with

contexts which distinguish them:

distinguished terms		*distinguishing context*
true	false	$\mathsf{if}\ \_\ \mathsf{then}\ \delta\ \mathsf{else}\ [*]$
δ	$[\delta]$	$\mathsf{let}\, x \Leftarrow \_\ \mathsf{in}\ [*]$
$[[\mathsf{true}]] \,\square\, [[\mathsf{false}]]$	$[[\mathsf{true}]] \,\square\, [\mathsf{false}]$	$\mathsf{let}\, w \Leftarrow \_\ \mathsf{in}\ \mathsf{let}\, x \Leftarrow w\ \mathsf{in}\ \mathsf{let}\, y \Leftarrow w\ \mathsf{in}$ $\mathsf{let}\, z \Leftarrow \mathsf{eq}(x,y)\ \mathsf{in}\ \mathsf{if}\, z\, \mathsf{then}\, \delta\, \mathsf{else}\, [*]$

In each case the first term fails the test and the second passes.

4.3 Bisimulation

We would like to show that (up to may testing) NMML satisfies the equational properties of $\mathsf{MML}\lambda\mathsf{bd}\Sigma$ used in Section 3, since this would tell us that programs viewed up to may testing form a cccc with computational coproducts of 1.

Unfortunately, proving equational properties are true for may-testing is quite difficult, because it requires quantifying over all contexts. For this reason, we will investigate *bisimulation* as an equivalence between programs, since it is much simpler to show the required equations are true for bisimulation, and then to show that bisimulation is finer than may testing.

In this section we shall define a variant of Milner's (1989) bisimulation for $\mathsf{NMML}\Sigma$, show that bisimulation is finer than may-testing, and that bsimulation satisfies the equational properties of $\mathsf{MML}\lambda\mathsf{bd}\Sigma$. Thus NMML fits the framework outlined in Section 3.

In this section, we shall use the theory of bisimulation for higher-order terms, first suggested by Abramsky (1989), adapted for a small-step labelled transition system. This section owes a great deal to Gordon's (1995) theory of bisimulation for functional languages, and to Howe's (1989) presentation of bisimulation for functional languages.

A family of relations $\mathcal{R}$ is *closed type-indexed* iff for each type τ, there is a relation $\mathcal{R}_\tau \subseteq \{(e,f) \mid \vdash e, f : \tau\}$.

A family of relations $\mathcal{R}$ is *open type-indexed* iff for each context Γ and type τ there is a relation $\mathcal{R}_{\Gamma,\tau} \subseteq \{(e,f) \mid \Gamma \vdash e, f : \tau\}$.

Given a closed type-indexed relation $\mathcal{R}$, let $\mathcal{R}^\circ$ be the open type-indexed relation given by:

$$\mathcal{R}^\circ_{\vec{x}:\vec{\sigma},\tau} = \{(e,f) \mid \forall \vdash \vec{g} : \vec{\sigma}\,.\, e[\vec{g}/\vec{x}]\ \mathcal{R}_\tau\ f[\vec{g}/\vec{x}]\}$$

Given a closed type-indexed relation $\mathcal{R}$, let $[\mathcal{R}]$ be the largest closed type-indexed relation such that:

- if $e\ [\mathcal{R}]_{[A]}\ f$ then $e = f$.
- if $(e,e')\ [\mathcal{R}]_{\sigma\otimes\tau}\ (f,f')$ then $e\ \mathcal{R}_\sigma\ f$ and $e'\ \mathcal{R}_\tau\ f'$,
- if $(\lambda x\,.\,e)\ [\mathcal{R}]_{\sigma\to \mathsf{C}\tau}\ (\lambda y\,.\,f)$ then for all $\vdash g : \sigma$ we have $e[g/x]\ \mathcal{R}_{\mathsf{C}\tau}\ f[g/y]$,
- if $e\ [\mathcal{R}]_{\mathsf{C}\tau}\ f$ and $e \xrightarrow{\iota} e'$ then $f \Longrightarrow f'$ and $e'\ \mathcal{R}_{\mathsf{C}\tau}\ f'$, and
- if $e\ [\mathcal{R}]_{\mathsf{C}\tau}\ f$ and $e \xrightarrow{\surd} e'$ then $f \overset{\surd}{\Longrightarrow} f'$ and $e'\ \mathcal{R}_\tau\ f'$.

A (*higher order weak*) *simulation* on $\mathsf{NMML}\Sigma$ is a closed type-indexed relation $\mathcal{R}$ such that $[\mathcal{R}] \subseteq \mathcal{R}$. A *bisimulation* is a simulation whose inverse is also a simulation. Then

$$
\begin{aligned}
x &\approx^\circ * \\
(v.\mathrm{L}, v.\mathrm{R}) &\approx^\circ v \\
\mathsf{let}\, x \Leftarrow [e]\, \mathsf{in}\, f &\approx^\circ f[e/x] \\
\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, [x] &\approx^\circ e \\
\mathsf{let}\, y \Leftarrow (\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f)\, \mathsf{in}\, g &\approx^\circ \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, (\mathsf{let}\, y \Leftarrow f\, \mathsf{in}\, g) \\
(\lambda x\,.\,e)\, f &\approx^\circ e[f/x] \\
\lambda y\,.\,(g y) &\approx^\circ g \\
\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, f\, \mathsf{else}\, g &\approx^\circ f \\
\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, f\, \mathsf{else}\, g &\approx^\circ g \\
\mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, f &\approx^\circ f \\
\mathsf{if}\, x\, \mathsf{then}\, f[\mathsf{true}/x]\, \mathsf{else}\, g[\mathsf{false}/x] &\approx^\circ \mathsf{if}\, x\, \mathsf{then}\, f\, \mathsf{else}\, g
\end{aligned}
$$

TABLE 1. Equations for $\mathsf{MML}\lambda\mathsf{bd}\Sigma$ expressed as bisimulations (y not free in g)

define:

- *simulation preorder* $\preceq$ is the largest simulation,
- *mutual simulation equivalence* $\asymp$ is $\preceq \cap \succeq$.
- *bisimulation equivalence* $\approx$ is the largest bisimulation.

Note that these are well-defined because $[\_]$ is monotone. We shall often elide the indices from these relations, writing $e\ \mathcal{R}\ f$ rather than $e\ \mathcal{R}_\tau\ f$ and $e\ \mathcal{R}^\circ\ f$ for $e\ \mathcal{R}^\circ_{\Gamma,\tau}\ f$ when context makes the typing obvious.

Note that bisimulation is strictly finer then mutual simulation, for example:

$$[*] \sqcap \delta \asymp [*] \qquad [*] \sqcap \delta \not\approx [*]$$

As this example shows, mutual simulation does not have the power to detect deadlock, which is why Milner (1989, exercise 9.14) chose to use bisimulation rather than mutual simulation for CCS.

We can show that (up to bisimulation) $\mathsf{NMML}\Sigma$ satisfies the equations of $\mathsf{MML}\lambda\mathsf{bd}$, by establishing bisimulations for the equations in Table 1. Thus, if we can show that bisimulation is finer than may-testing, we have shown that (up to may-testing) $\mathsf{NMML}\Sigma$ satisfies the equations of $\mathsf{MML}\lambda\mathsf{bd}$. This is trivial to establish *if* we can show that bisimulation is a congruence, which is what the rest of this section will show.

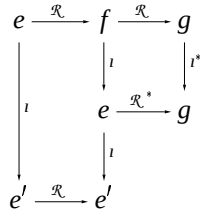
Unfortunately, it is quite tricky to show that bisimulation is a congruence, since the direct proof based on Milner's (1989) proof for CCS fails in the higher-order case. It is routine to show directly that the relation:

$$\mathcal{R} = \{(C[e], C[f]) \mid e \approx f\}$$

satisfies the property:

- If $e\ \mathcal{R}\ f$ and $e \xrightarrow{\iota} e'$ then $f \Longrightarrow f'$ and $e'\ \mathcal{R}^*\ f'$.

However, this is *not* enough to show that $\mathcal{R}^*$ is a simulation, for reasons similar to the problems with showing that the λ-calculus is Church–Rosser, since the above property allows for systems such as:



where it would be impossible to close the diagram.

We shall now follow a variant of Gordon's (1995) presentation of Howe's (1989) proof that simulation is a precongruence. Define the *one-level deep* contexts to be:

$$\begin{aligned} D[e_1,\ldots,e_n] = x \mid * \mid (e_1,e_2) \mid c(e_1,\ldots,e_n) \mid d(e_1,\ldots,e_n) \\ \mid [e_1] \mid \mathsf{let}\, x \Leftarrow e_1 \,\mathsf{in}\, e_2 \\ \mid \lambda x\,.\,e_1 \mid e_1 e_2 \\ \mid \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 \\ \mid \delta \mid e_1 \,\square\, e_2 \mid \mathsf{fix}(x = e_1) \end{aligned}$$

Note that for any e there is a unique D and $\vec{e}$ such that $e = D[\vec{e}]$. Given an open type-indexed relation $\mathcal{R}$, let $\widehat{\mathcal{R}}$ be given by:

$$\widehat{\mathcal{R}} = \{(D[\vec{e}], D[\vec{f}]) \mid \vec{e}\;\mathcal{R}\;\vec{f}\}$$

Given a closed type-indexed relation $\mathcal{R}$, let $\mathcal{R}^\bullet$ be the *compatible closure* of $\mathcal{R}$ given by:

$$\frac{e\;\widehat{\mathcal{R}^\bullet}\;f\;\mathcal{R}^\circ\;g}{e\;\mathcal{R}^\bullet\;g}$$

PROPOSITION 9. *For any preorder $\leq$:*

1. $\leq^\bullet \leq^\circ \subseteq \leq^\bullet$
2. $\leq^\bullet$ *is reflexive.*
3. $\leq^\circ \subseteq \leq^\bullet$.
4. *If* $e \leq^\bullet f$ *and* $e' \leq^\bullet f'$ *then* $e[e'/x] \leq^\bullet f[f'/x]$.

PROOF.

1. If $e \leq^\bullet f \leq^\circ g$ then $e\;\widehat{\leq^\bullet}\;h \leq^\circ f \leq^\circ g$, and so $e \leq^\bullet g$.
2. For any e, show by induction on e that $e \leq^\bullet e$.
3. Since $\leq^\bullet$ is reflexive, $\widehat{\leq^\bullet}$ is reflexive, and so $\leq^\circ \subseteq \widehat{\leq^\bullet} \leq^\circ \subseteq \leq^\bullet$.
4. An induction on e. □

PROPOSITION 10. *If $\mathcal{R}$ is a simulation and a preorder, then $\mathcal{R}^\bullet$ is a simulation.*

PROOF. For any closed e, show by induction on e that if $e\ \mathcal{R}^\bullet\ f$ then $e\ [\mathcal{R}^\bullet]\ f$. We shall prove the case when $e = \mathsf{let}\, x \Leftarrow e_1 \,\mathsf{in}\, e_2$, and the other cases are similar. Since $e\ \mathcal{R}^\bullet\ f$ we can find $g = \mathsf{let}\, x \Leftarrow g_1 \,\mathsf{in}\, g_2$ such that $e_i\ \mathcal{R}^\bullet\ g_i$ and $g\ \mathcal{R}\ f$. Then if $e \xrightarrow{\iota} e'$ then we have two cases:

- either $e_1 \xrightarrow{\iota} e_1'$ and $e' = \mathsf{let}\, x \Leftarrow e_1' \,\mathsf{in}\, e_2$, so by induction $g_1 \Longrightarrow g_1'$ and $e_1'\ \mathcal{R}^\bullet\ g_1'$, so $g \Longrightarrow g' = \mathsf{let}\, x \Leftarrow g_1' \,\mathsf{in}\, g_2$ and $e'\ \mathcal{R}^\bullet\ g'$, so $f \Longrightarrow f'$ and $e'\ \mathcal{R}^\bullet\ g'\ \mathcal{R}\ f'$, and so $e'\ \mathcal{R}^\bullet\ f'$,
- or $e_1 \xrightarrow{\surd} e_1'$ and $e' = e_2[e_1'/x]$, so by induction $g_1 \stackrel{\surd}{\Longrightarrow} g_1'$ and $e_1'\ \mathcal{R}^\bullet\ g_1'$, so $g \Longrightarrow g' = g_2[g_1'/x]$ and $e'\ \mathcal{R}^\bullet\ g'$, so $f \Longrightarrow f'$ and $g'\ \mathcal{R}\ f'$, and so $e'\ \mathcal{R}^\bullet\ f'$,

Thus $\mathcal{R}^\bullet \subseteq [\mathcal{R}^\bullet]$, and so $\mathcal{R}^\bullet$ is a simulation. □

Proposition 10 is sufficient to show that simulation is a precongruence, which is the result shown by Howe. To show that bisimulation is a congruence, we need the following unpublished observation of Howe's (1992) pointed out to the author by Andrew Pitts:

PROPOSITION 11. *If $\mathcal{R}$ is symmetric then so is $\mathcal{R}^{\bullet *}$.*

PROOF. Show by induction on e that if $e\ \mathcal{R}^\bullet\ f$ then $f\,\mathcal{R}^{\bullet *}\,e$. From this it is routine to establish that $\mathcal{R}^{\bullet *}$ is symmetric. □

We can then plug Propositions 10 and 11 together to show that $\approx^\circ$ is a congruence.

PROPOSITION 12. *$\approx^\circ$ is a congruence.*

PROOF. By Proposition 9, $\approx^\circ \subseteq \approx^\bullet$ and $\approx^\bullet$ is a congruence. By Proposition 10 $\approx^\bullet$ is a simulation, and so $\approx^{\bullet *}$ is a simulation. By Proposition 11 $\approx^{\bullet *}$ is symmetric, so $\approx^{\bullet *}$ is a bisimulation, and so $\approx^\bullet \subseteq \approx^{\bullet *} \subseteq \approx^\circ$. Thus $\approx^\circ = \approx^\bullet$ is a congruence. □

Having shown that $\approx^\circ$ is a precongruence, we can show that bisimulation is finer than may-testing.

PROPOSITION 13. *If $e \approx^\circ_{\Gamma,\tau} f$ then $\Gamma \models e \sqsubseteq_O f$.*

PROOF. For any closing context C of type $\mathsf{C}I$, by Proposition 12, $C[e] \approx^\circ C[f]$, so if $C[e] \stackrel{\surd}{\Longrightarrow} *$ then $C[f] \stackrel{\surd}{\Longrightarrow} *$. Thus $\Gamma \models e \sqsubseteq_O f$. □

Let $\mathsf{NMML}_\approx\Sigma$ be $\mathsf{NMML}\Sigma$ viewed up to bisimulation. It is routine to verify that $\mathsf{NMML}_\approx\Sigma$ forms a signature in the same way as $\mathsf{MML}\lambda\mathsf{bd}\Sigma$, and that $\mathsf{NMML}_\approx : \mathbf{SigBCD} \to \mathbf{SigBCD}$ is a monad.

PROPOSITION 14. *Any $\mathsf{NMML}_\approx$-algebra is a cccc with computational coproducts of* 1.

PROOF. $\mathsf{NMML}\Sigma$ satisfies the equations in Table 1, up to bisimulation, so $\mathsf{NMML}_\approx\Sigma$ is a $\mathsf{MML}\lambda\mathsf{bd}$-algebra, and so by Proposition 6 is a cccc with computational coproducts of 1. Thus, any $\mathsf{NMML}_\approx$-algebra must be a cccc with computational coproducts of 1. □

4.4 Denotational semantics

In this section we present a denotational semantics for NMML based on powerdomains. The rest of this section will show this semantics is fully abstract for may-testing.

In the previous section we saw that any $\mathsf{NMML}_{\approx}$-algebra must be a cccc with computational coproducts of 1. We will model NMML in a particular such cccc **Alg** with the lower powerdomain monad $\mathcal{P}$. This is a cccc with computational coproducts of 1, and so has a denotational semantics for $\mathsf{MML}\lambda\mathsf{bd}$ given by Propositions 2, 3, 4, 6 and 7.

The semantics for $\mathsf{NMML}\Sigma$ extends this with:

$$\begin{aligned}
[\![\Gamma \vdash \delta : \mathsf{C}\tau]\!] &= \bot \\
[\![\Gamma \vdash e \,\square\, f : \mathsf{C}\tau]\!] &= [\![\Gamma \vdash e : \mathsf{C}\tau]\!] \vee [\![\Gamma \vdash f : \mathsf{C}\tau]\!] \\
[\![\Gamma \vdash \mathsf{fix}(x = e) : \mathsf{C}\tau]\!] &= \text{the least fixed pt of } f \mapsto \langle \mathsf{id}, f \rangle ; [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!]
\end{aligned}$$

Note that this semantics is well-defined because $\mathcal{P}\mathbf{D}$ is a join semi-lattice.

Thus for any Σ, if there is a morphism $[\![\_]\!] : \Sigma \to \mathbf{Alg}$ then we can extend this to $[\![\_]\!] : \mathsf{NMML}\Sigma \to \mathbf{Alg}$ as:

$$\mathsf{NMML}\Sigma \xrightarrow{\mathsf{NMML}[\![\_]\!]} \mathsf{NMML}\mathbf{Alg} \xrightarrow{[\![\_]\!]} \mathbf{Alg}$$

For example, we have a mapping $[\![\_]\!] : \mathsf{NatList} \to \mathbf{Alg}$ given in Section 3.1, which maps the sorts of $\mathsf{NatList}$ to discrete domains:

$$[\![\mathsf{bool}]\!] = \{t, f\} \qquad [\![\mathsf{nat}]\!] = \omega \qquad [\![\mathsf{list}]\!] = \omega^*$$

the constructors to continuous functions (since every function between discrete domains is continuous):

$$\begin{aligned}
[\![\mathsf{true}]\!] &= t & [\![\mathsf{false}]\!] &= f \\
[\![\mathsf{zero}]\!] &= 0 & [\![\mathsf{succ}]\!] &= _ + 1 \\
[\![\mathsf{nil}]\!] &= \varepsilon & [\![\mathsf{cons}]\!] &= _._
\end{aligned}$$

and the deconstructors to continuous functions in the Kleisli category (that is functions $X \to \mathcal{P}Y$):

$$[\![\mathsf{pred}]\!] = n \mapsto \begin{cases} \{n-1\} & \text{if } n > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\mathsf{eq}]\!] = (m,n) \mapsto \begin{cases} \{t\} & \text{if } m = n \\ \{f\} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{isnil}]\!] = l \mapsto \begin{cases} \{t\} & \text{if } l = \varepsilon \\ \{f\} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{hd}]\!] = l \mapsto \begin{cases} \{n\} & \text{if } l = n.l' \\ \emptyset & \text{if} l = \varepsilon \end{cases}$$

$$[\![\mathsf{tl}]\!] = l \mapsto \begin{cases} \{l\}' & \text{if } l = n.l' \\ \emptyset & \text{if} l = \varepsilon \end{cases}$$

This means we have a semantics $[\![\_]\!] : \mathsf{NMML}\Sigma \to \mathbf{Alg}$, for example:

$$\begin{aligned}
[\![\vdash \mathsf{true} : [\mathsf{bool}]]\!] &= t \\
[\![\vdash \mathsf{false} : [\mathsf{bool}]]\!] &= f \\
[\![\vdash \delta : \mathsf{CC}\tau]\!] &= \emptyset
\end{aligned}$$

$$\begin{aligned}
[\![\vdash [\delta] : \mathsf{C}\,\mathsf{C}\tau]\!] &= \{\emptyset\} \\
[\![\vdash [[\mathsf{true}]] \,\square\, [[\mathsf{false}]] : \mathsf{C}\,\mathsf{C}[\mathsf{bool}]]\!] &= \{\{t\},\{f\}\} \\
[\![\vdash [[\mathsf{true}] \,\square\, [\mathsf{false}]] : \mathsf{C}\,\mathsf{C}[\mathsf{bool}]]\!] &= \{\{t\},\{f\},\{t,f\}\} \\
[\![\vdash \mathsf{snoc} : \mathsf{list} \otimes \mathsf{nat} \to \mathsf{C}\,\mathsf{list}]\!] &= (l,n) \mapsto \{l.n\}
\end{aligned}$$

Define the denotational preorder $\Gamma \vdash e \sqsubseteq_D f : \tau$ iff $[\![\Gamma \vdash e : \tau]\!] \leq [\![\Gamma \vdash f : \tau]\!]$.

A semantics $[\![\_]\!] : \Sigma \to \mathbf{Alg}$ is *adequate* iff:

$$[\![\vdash d\vec{e} : \mathsf{C}[A]]\!] = \bigvee \{ [\![\vdash [f] : \mathsf{C}[A]]\!] \mid d\vec{e} \stackrel{\checkmark}{\Longrightarrow} f \}$$

For example, the above semantics for $\mathsf{NatList}$ is adequate, as can be verified by comparing its operational and denotational semantics.

A semantics $[\![\_]\!] : \Sigma \to \mathbf{Alg}$ is *expressive* iff for any compact $a \in [\![A]\!]$ we can find terms is_a and test_a such that:

$$[\![\vdash \mathsf{is}_a : [A]]\!] = a \qquad [\![\vdash \mathsf{test}_a : [A] \to \mathsf{C}\,I]\!] = (a \Rightarrow \eta\bot)$$

For example, the above semantics for $\mathsf{NatList}$ is expressive, since we can define:

$$\begin{aligned}
\mathsf{is}_t &= \mathsf{true} \\
\mathsf{is}_f &= \mathsf{false} \\
\mathsf{is}_0 &= \mathsf{zero} \\
\mathsf{is}_{n+1} &= \mathsf{succ}\,\mathsf{is}_n \\
\mathsf{is}_\varepsilon &= \mathsf{nil} \\
\mathsf{is}_{n.l} &= \mathsf{cons}(\mathsf{is}_n, \mathsf{is}_l) \\
\mathsf{test}_t &= \lambda x\,.\,\mathsf{if}\,x\,\mathsf{then}\,[*]\,\mathsf{else}\,\delta \\
\mathsf{test}_f &= \lambda x\,.\,\mathsf{if}\,x\,\mathsf{then}\,\delta\,\mathsf{else}\,[*] \\
\mathsf{test}_n &= \lambda x\,.\,\mathsf{let}\,y \Leftarrow \mathsf{eq}(x, \mathsf{is}_n)\,\mathsf{in}\,\mathsf{test}_t y \\
\mathsf{test}_\varepsilon &= \lambda x\,.\,\mathsf{let}\,y \Leftarrow \mathsf{isnil}\,x\,\mathsf{in}\,\mathsf{test}_t y \\
\mathsf{test}_{n.l} &= \lambda w\,.\,\mathsf{let}\,x \Leftarrow \mathsf{hd}\,w\,\mathsf{in}\,\mathsf{let}\,y \Leftarrow \mathsf{test}_n x\,\mathsf{in}\,\mathsf{let}\,z \Leftarrow \mathsf{tl}\,w\,\mathsf{in}\,\mathsf{test}_l z
\end{aligned}$$

Moreover, any signature with equality sorts automatically has an adequate and expressive semantics:

PROPOSITION 15. *Any signature with all sorts being equality sorts has an adequate expressive semantics in* **Alg**.

PROOF. Let $[\![A]\!]$ be the discrete poset of terms:

$$\begin{aligned}
[\![A]\!] &= \{e \mid \vdash e : [A]\} \\
[\![c]\!] &= \vec{e} \mapsto c\vec{e} \\
[\![d]\!] &= \vec{e} \mapsto \{f \mid d\vec{e} \stackrel{\checkmark}{\Longrightarrow} f\}
\end{aligned}$$

This is adequate, and we can define:

$$\mathsf{is}_e = e \qquad \mathsf{test}_e = \lambda x\,.\,\mathsf{let}\,y \Leftarrow \mathsf{eq}(x,e)\,\mathsf{in}\,\mathsf{if}\,x\,\mathsf{then}\,[*]\,\mathsf{else}\,\delta$$

and verify that the signature is expressive. □

A semantics $[\![\_]\!] : \mathsf{NMML}\Sigma \to \mathbf{Alg}$ is *correct* iff:

$$[\![\Gamma \vdash e : \tau]\!] \leq [\![\Gamma \vdash f : \tau]\!] \text{ implies } \Gamma \models e \sqsubseteq_O f : \tau$$

The semantics for $\mathsf{NMML}\Sigma$ is *fully abstract* iff this can be strengthened to:

$$[\![\Gamma \vdash e : \tau]\!] \leq [\![\Gamma \vdash f : \tau]\!] \text{ iff } \Gamma \models e \sqsubseteq_O f : \tau$$

The rest of this section will show that if a semantics for Σ is adequate then its extension to $\mathsf{NMML}\Sigma$ is correct, and that if a semantics for Σ is adequate and expressive, then its extension to $\mathsf{NMML}\Sigma$ is fully abstract. In particular Proposition 15 means that if Σ consists of equality sorts, then $\mathsf{NMML}\Sigma$ has a fully abstract semantics.

4.5 Program logic

In order to show the relationship between the operational and denotational semantics of $\mathsf{NMML}\Sigma$, we shall use a *program logic* similar to that used by Abramsky (1989) and Ong (1988) in modelling the untyped λ-calculus, based on Abramsky's (1991) *domain theory in logical form*.

The logic is presented in two ways:

- it has an *operational* characterization, similar to the operational characterization for HML (Milner, 1989) or the modal μ-calculus (Kozen, 1983), and
- it has a *denotational* characterization, which provides a syntax for the compact elements of $[\![\tau]\!]$, in a similar fashion to Scott's (1982) *information systems*.

In Section 4.6 we shall see a third presentation of the logic, using *sequent calculus*. In Section 4.8 we shall show that these three presentations are equivalent, and use this to show full abstraction for the powerdomain semantics for $\mathsf{NMML}\Sigma$.

The *program logic* for $\mathsf{NMML}\Sigma$ has propositions:

$$\phi ::= * \mid (\phi, \psi) \mid |a| \mid \omega \mid \phi \wedge \psi \mid [\phi] \mid \phi \Rightarrow \psi$$

These can be statically typed, so the propositions for type τ are those where $\phi : \mathcal{L}\tau$:

$$\frac{}{* : \mathcal{L}I} \quad \frac{\phi : \mathcal{L}\sigma \quad \psi : \mathcal{L}\tau}{(\phi,\psi) : \mathcal{L}(\sigma \otimes \tau)} \quad \frac{}{|a| : \mathcal{L}[A]}[a \in [\![A]\!], a \text{ is compact}]$$

$$\frac{}{\omega : \mathcal{L}(\mathsf{C}\tau)} \quad \frac{\phi : \mathcal{L}(\mathsf{C}\tau) \quad \psi : \mathcal{L}(\mathsf{C}\tau)}{\phi \wedge \psi : \mathcal{L}(\mathsf{C}\tau)} \quad \frac{\phi : \mathcal{L}\tau}{[\phi] : \mathcal{L}(\mathsf{C}\tau)}$$

$$\frac{}{\omega : \mathcal{L}(\sigma \to \mathsf{C}\tau)} \quad \frac{\phi : \mathcal{L}(\sigma \to \mathsf{C}\tau) \quad \psi : \mathcal{L}(\sigma \to \mathsf{C}\tau)}{\phi \wedge \psi : \mathcal{L}(\sigma \to \mathsf{C}\tau)} \quad \frac{\phi : \mathcal{L}\sigma \quad \psi : \mathcal{L}(\mathsf{C}\tau)}{\phi \Rightarrow \psi : \mathcal{L}(\sigma \to \mathsf{C}\tau)}$$

We can give an informal account of these propositions as:

- any term $\vdash e : I$ satisfies $*$,
- $\vdash (e, f) : \sigma \otimes \tau$ satisfies (ϕ, ψ) iff e satisfies ϕ and f satisfies ψ,
- $\vdash e : [A]$ satisfies $|a|$ iff $[\![e]\!] \leq a$,
- $\vdash e : \mathsf{C}\tau$ satisfies $[\phi]$ iff e can terminate with some result f which satisfies ϕ,
- $\vdash e : \sigma \to \mathsf{C}\tau$ satisfies $\phi \Rightarrow \psi$ iff whenever f satisfies ϕ then $e\,f$ satisfies ψ,

- any term satisfies ω, and
- a term satisfies $\phi \wedge \psi$ iff it satisfies ϕ and ψ.

For example, some terms and predicates they satisfy are:

$$\begin{aligned}
\mathsf{true} &\text{ satisfies } |t| \\
\mathsf{succ}\,\mathsf{succ}\,\mathsf{zero} &\text{ satisfies } |2| \\
\delta &\text{ satisfies } \omega \\
[\delta] &\text{ satisfies } [\omega] \\
[[\mathsf{true}]] \,\square\, [[\mathsf{false}]] &\text{ satisfies } [[|t|]] \wedge [[|f|]] \\
[[\mathsf{true}] \,\square\, [\mathsf{false}]] &\text{ satisfies } [[|t|] \wedge [|f|]] \\
\mathsf{snoc} &\text{ satisfies } (|0.1|, |2|) \Rightarrow [|0.1.2|]
\end{aligned}$$

Note that this logic includes compact elements of base type as formulae. It is possible to replace this by requiring the base types to have a program logic characterizing the compacts, but this would unnecessarily complicate the proofs.

We can formalize this notion of when a term satisfies a proposition by giving the operational characterization of the logic as judgements $\models e : \phi$ given by:

$$\frac{}{\models * : *} \qquad \frac{\models e : \phi \quad \models f : \psi}{\models (e,f) : (\phi,\psi)} \qquad \frac{a \leq [\![\vdash e : [A]]\!]}{\models e : |a|}$$

$$\frac{}{\models e : \omega} \qquad \frac{\models e : \phi \quad \models e : \psi}{\models e : \phi \wedge \psi} \qquad \frac{e \xrightarrow{1} e' \quad \models e' : \phi}{\models e : \phi} \qquad \frac{e \xrightarrow{\checkmark} f \quad \models f : \phi}{\models f : [\phi]}$$

$$\frac{\forall \models f : \phi \,.\, \models ef : \psi}{\models e : \phi \Rightarrow \psi}$$

This can be generalized to open terms as:

$$\vec{x} : \vec{\phi} \models e : \psi \text{ iff } \forall \models \vec{f} : \vec{\phi} \,.\, \models e[\vec{f}/\vec{x}] : \psi$$

For example:

$$\begin{aligned}
x : \phi, y : \psi &\models (x,y) : (\phi,\psi) \\
x : \phi &\models [x] : [\phi] \\
x : (|0.1|, |2|) &\models \mathsf{snoc}\, x : [|0.1.2|]
\end{aligned}$$

Let Δ range over propositional contexts of the form $x_1 : \phi_1, \ldots, x_n : \phi_n$, and write $\Delta : \mathcal{L}\Gamma$ for:

$$(x_1 : \phi_1, \ldots, x_n : \phi_n) : \mathcal{L}(x_1 : \tau_1, \ldots, x_n : \tau_n) \text{ iff } \phi_1 : \mathcal{L}\tau_1, \ldots, \phi_n : \mathcal{L}\tau_n$$

We can also define a denotational semantics for propositions, so that if $\phi : \mathcal{L}\tau$ then $[\![\phi]\!] \in [\![\tau]\!]$:

$$\begin{aligned}
[\![*]\!] &= \bot \\
[\![(\phi,\psi)]\!] &= ([\![\phi]\!], [\![\psi]\!]) \\
[\![|a|]\!] &= a \\
[\![\omega]\!] &= \bot
\end{aligned}$$

$$[\![\phi \wedge \psi]\!] = [\![\phi]\!] \vee [\![\psi]\!]$$
$$[\![[\phi]]\!] = \eta[\![\phi]\!]$$
$$[\![\phi \Rightarrow \psi]\!] = [\![\phi]\!] \Rightarrow [\![\psi]\!]$$

This gives us a syntax for compact elements of $[\![\tau]\!]$, since we can show that the compact elements of $[\![\tau]\!]$ are precisely the denotations of propositions from $\mathcal{L}\tau$.

PROPOSITION 16. *$a \in [\![\tau]\!]$ is compact iff $\exists \phi : \mathcal{L}\tau \,.\, a = [\![\phi]\!]$.*

PROOF. $\Rightarrow$ is an induction on τ. $\Leftarrow$ is an induction on ϕ. □

Whenever $\Delta : \mathcal{L}\Gamma$, we can define $[\![\Delta]\!] \in [\![\Gamma]\!]$ as:

$$[\![x_1 : \phi_1, \ldots, x_n : \phi_n]\!] = ([\![\phi_1]\!], \ldots, [\![\phi_n]\!])$$

Then in Section 4.8 we shall see that these two presentations of the logic are equivalent, in that whenever $\Gamma \vdash e : \tau$, $\Delta : \mathcal{L}\Gamma$ and $\phi : \mathcal{L}\tau$ we have:

$$\Delta \models e : \phi \text{ iff } [\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]$$

This result is an important step in proving the semantics fully abstract.

For those readers familiar with Abramsky's (1991) domain theory in logical form, it is worth noting some differences between his logic and ours. Abramsky's logic allows finite conjunction (using ω and $\wedge$) for propositions of any type, not just $\mathcal{L}(\mathsf{C}\tau)$ and $\mathcal{L}(\sigma \to \mathsf{C}\tau)$. Thus, propositions in Abramsky's logic do not represent compact elements (as ours do) but represent compact Scott-open sets of elements. This allows Abramsky to use the theory of *Stone duality* (Johnstone, 1982, for example) in showing the relationship between program logics and denotational semantics.

However, for the proofs given here, it is simpler to restrict the use of conjunction to propositions of type $\mathcal{L}(\mathsf{C}\tau)$ and $\mathcal{L}(\sigma \to \mathsf{C}\tau)$, whose semantics in **Alg** form join semilattices. This allows us to use propositions as a syntactic representation of compacts, and simplifies some of the proofs in later sections.

It is an open question as to what the relationship between these two logics is. One possibility is that Abramsky's logic can be seen as representing compact morphisms in the Kleisli category $\mathbf{Alg}_{\mathcal{P}}$ where ours represent compact morphisms in the underlying category **Alg**. We will not investigate this possibility further here.

4.6 Proof system

In order to relate the denotational and operational characterizations of the program logic, we shall use an intermediate proof system. This is a sequent calculus with judgements of the form $\Delta \vdash e : \phi$ where $\Gamma \vdash e : \tau$, $\Delta : \mathcal{L}\Gamma$ and $\phi : \mathcal{L}\tau$. In this section we shall define this proof system, and show that $\Delta \vdash e : \phi$ iff $[\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]$.

To begin with, we give a complete axiomatization for the semantics of the program logic. Let $\leq$ be the preorder on propositions given by:

- ω is the top element, and $(\_ \wedge \_)$ is meet.
- $(\_, \_)$, $[\_]$ and $(\phi \Rightarrow \_)$ are monotone.
- $(\phi \Rightarrow \_)$ preserves ω and $\wedge$.

- $|\_|$ and $(\_ \Rightarrow \psi)$ are anti-monotone.

PROPOSITION 17. $\phi \leq \psi$ *iff* $[\![\phi]\!] \geq [\![\psi]\!]$.

PROOF. $\Rightarrow$ is an induction on τ. $\Leftarrow$ is an induction on ϕ. □

We can then define the proof system for $\mathsf{NMML}\Sigma$ as:

$$\frac{[\![\phi]\!] \leq [\![\Gamma \vdash c\vec{e} : [A]]\!][\![\Delta]\!]}{\Delta \vdash c\vec{e} : \phi} \quad \frac{[\![\phi]\!] \leq [\![\Gamma \vdash d\vec{e} : \mathsf{C}[A]]\!][\![\Delta]\!]}{\Delta \vdash d\vec{e} : \phi} \quad \frac{\Delta \vdash e : \psi}{\Delta \vdash e : \phi}[\psi \leq \phi]$$

$$\frac{}{\Delta, x : \phi \vdash x : \phi} \quad \frac{\Delta \vdash x : \phi}{\Delta, y : \psi \vdash x : \phi}[x \neq y]$$

$$\frac{}{\Delta \vdash * : *} \quad \frac{\Delta \vdash e : \phi \quad \Delta \vdash f : \psi}{\Delta \vdash (e, f) : (\phi, \psi)}$$

$$\frac{}{\Delta \vdash e : \omega} \quad \frac{\Delta \vdash e : \phi \quad \Delta \vdash e : \psi}{\Delta \vdash e : \phi \wedge \psi}$$

$$\frac{\Delta \vdash e : \phi}{\Delta \vdash [e] : [\phi]} \quad \frac{\Delta \vdash e : [\phi] \quad \Delta, x : \phi \vdash f : \psi}{\Delta \vdash \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f : \psi}$$

$$\frac{\Delta, x : \psi \vdash e : \chi}{\Delta \vdash \lambda x\,.\,e : \psi \Rightarrow \chi} \quad \frac{\Delta \vdash e : \psi \Rightarrow \chi \quad \Delta \vdash f : \psi}{\Delta \vdash ef : \chi}$$

$$\frac{\Delta \vdash e : |t| \quad \Delta \vdash f : \phi}{\Delta \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : \phi} \quad \frac{\Delta \vdash e : |f| \quad \Delta \vdash g : \phi}{\Delta \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : \phi}$$

$$\frac{\Delta \vdash e : \psi \quad \Delta \vdash f : \chi}{\Delta \vdash e \,\Box\, f : \psi \wedge \chi} \quad \frac{\Delta \vdash \mathsf{fix}(x = e) : \psi \quad \Delta, x : \psi \vdash e : \chi}{\Delta \vdash \mathsf{fix}(x = e) : \chi}$$

Note that all of the structural rules for the proof system, such as weakening and contraction, have been absorbed into the definition of $\phi \leq \psi$.

PROPOSITION 18. $\Delta \vdash e : \phi$ *iff* $[\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]$.

PROOF.

$\Rightarrow$ An induction on the proof of $\Delta \vdash e : \phi$.

$\Leftarrow$ An induction on the proof of $\Gamma \vdash e : \tau$. The tricky case is $\Gamma \vdash \mathsf{fix}(x = e) : \mathsf{C}\tau$, in which case we have:

$$[\![\phi]\!] \leq \bigvee_n (f \mapsto \langle \mathsf{id}, f \rangle; [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!])^n \bot [\![\Delta]\!]$$

Since $[\![\phi]\!]$ is compact, we can find n such that:

$$[\![\phi]\!] \leq (f \mapsto \langle \mathsf{id}, f \rangle; [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!])^n \bot [\![\Delta]\!]$$

and we proceed by induction on n.

If $n = 0$ then $[\![\phi]\!] \leq \bot$, so $\vdash \phi \geq \omega : \mathcal{L}(\mathsf{C}\tau)$ and so $\Delta \vdash \mathsf{fix}(x = e) : \phi$.

If $n > 0$ then:

$$[\![\phi]\!] \leq [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!]([\![\Delta]\!], (f \mapsto \langle \mathsf{id}, f \rangle; [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!])^{n-1} \bot [\![\Delta]\!])$$

and so from Proposition 16 we can find ψ such that:

$$[\![\psi]\!] \leq (f \mapsto \langle \mathsf{id}, f\rangle; [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!])^{n-1} \bot [\![\Delta]\!]$$
$$[\![\phi]\!] \leq [\![\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau]\!]([\![\Delta]\!], [\![\psi]\!])$$

so by induction on the proof of $\Gamma \vdash \mathsf{fix}(x = e) : \mathsf{C}\tau$ and n we have:

$$\Delta \vdash \mathsf{fix}(x = e) : \psi \qquad \Delta, x : \psi \vdash e : \phi$$

and so $\Delta \vdash \mathsf{fix}(x = e) : \phi$. □

4.7 Expressivity

In this section we shall show that as long as Σ is expressive, then so is $\mathsf{NMML}\Sigma$, since for any $\phi : \mathcal{L}\tau$ we can define a term $\mathsf{term}_\tau \phi$ such that:

$$[\![\phi]\!] = [\![\vdash \mathsf{term}_\tau \phi : \tau]\!]$$

In particular, this means that for any proposition $\phi : \mathcal{L}\tau$, there is a context which determines whether a term $\vdash e : \tau$ satisfies that proposition:

$$[\![\vdash \mathsf{term}_\tau(\phi \Rightarrow [*])e : \mathsf{C}I]\!] = \begin{cases} \eta\bot \text{ if } [\![\phi]\!] \leq [\![\vdash e : \tau]\!] \\ \bot \quad \text{otherwise} \end{cases}$$

This expressivity result is used in showing that the semantics for $\mathsf{NMML}\Sigma$ is fully abstract. The relationship between expressivity and full abstraction has been long known (Plotkin, 1977, for example).

PROPOSITION 19. *If the semantics for Σ is expressive, then for any $\phi : \mathcal{L}\tau$ we can find $\vdash \mathsf{term}_\tau \phi : \tau$ such that $[\![\phi]\!] = [\![\vdash \mathsf{term}_\tau \phi : \tau]\!]$.*

PROOF. Let $\mathsf{term}_\tau \phi$ be defined:

$$\begin{aligned}
\mathsf{term}_I * &= * \\
\mathsf{term}_{\sigma\otimes\tau}(\phi, \psi) &= (\mathsf{term}_\sigma \phi, \mathit{term}_\tau \psi) \\
\mathsf{term}_{[A]}\,|a| &= \mathsf{is}_a \\
\mathsf{term}_{\mathsf{C}\tau}\,\omega &= \delta \\
\mathsf{term}_{\mathsf{C}\tau}(\phi \wedge \psi) &= \mathsf{term}_{\mathsf{C}\tau}\phi \,\Box\, \mathsf{term}_{\mathsf{C}\tau}\psi \\
\mathsf{term}_{\mathsf{C}\tau}[\phi] &= [\mathsf{term}_\tau \phi] \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}\,\omega &= \lambda x\,.\,\delta \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\phi \wedge \psi) &= \lambda x\,.\,(\mathsf{term}_{\sigma\to\mathsf{C}\tau}\phi)x \,\Box\, (\mathsf{term}_{\sigma\to\mathsf{C}\tau}\psi)x \\
\mathsf{term}_{I\to\mathsf{C}\tau}(* \Rightarrow \chi) &= \lambda x\,.\,\mathsf{term}_{\mathsf{C}\tau}\chi \\
\mathsf{term}_{\rho\otimes\sigma\to\mathsf{C}\tau}((\psi, \phi) \Rightarrow \chi) &= \lambda x\,.\,\mathsf{let}\, y \Leftarrow (\mathsf{term}_{\rho\to\mathsf{C}I}(\psi \Rightarrow [*]))(x.\mathrm{L}) \\
&\qquad \mathsf{in}(\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\phi \Rightarrow \chi))(x.\mathrm{R}) \\
\mathsf{term}_{[A]\to\mathsf{C}\tau}(|a| \Rightarrow \chi) &= \lambda x\,.\,\mathsf{let}\, y \Leftarrow (\mathsf{test}_a x)\,\mathsf{in}\,\mathsf{term}_{\mathsf{C}\tau}\chi \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\omega \Rightarrow \chi) &= \lambda x\,.\,\mathsf{term}_{\mathsf{C}\tau}\chi \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\phi \wedge \psi \Rightarrow \chi) &= \lambda x\,.\,\mathsf{let}\, y \Leftarrow \mathsf{term}_{\sigma\to\mathsf{C}I}(\phi \Rightarrow [*])x \\
&\qquad \mathsf{in}\,\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\psi \Rightarrow \chi)x
\end{aligned}$$

$$\mathsf{term}_{\mathsf{C}\sigma\to\mathsf{C}\tau}([\phi]\Rightarrow\chi) = \lambda x\,.\,\mathsf{let}\,y \Leftarrow x\,\mathsf{in}\,\mathsf{term}_{\sigma\to\mathsf{C}\tau}\,y$$
$$\mathsf{term}_{(\rho\to\mathsf{C}\sigma)\to\mathsf{C}\tau}((\phi\Rightarrow\psi)\Rightarrow\chi) = \lambda x\,.\,(\mathsf{term}_{\mathsf{C}\sigma\to\mathsf{C}\tau}(\psi\Rightarrow\chi))(x(\mathsf{term}_{\rho}\,\phi))$$

It is routine to verify that $[\![\phi]\!] = [\![\vdash \mathsf{term}_\tau\,\phi : \tau]\!]$. □

We can extend this result to contexts, to relate propositional contexts Δ to their program context equivalent C_Δ.

PROPOSITION 20. *If Σ is expressive then for any $\Delta : \measuredangle\Gamma$ there is a context C_Δ such that $[\![\Gamma \vdash e : \mathsf{C}\tau]\!][\![\Delta]\!] = [\![\Gamma \vdash C_\Delta[e] : \mathsf{C}\tau]\!]\bot$.*

PROOF. If $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ then let C_Δ be the context:

$$C_{x_1:\phi_1,\ldots,x_n:\phi_n}[e] = \mathsf{let}\,x_1 \Leftarrow [\mathsf{term}_{\tau_1}\,\phi_1]\,\mathsf{in}\cdots\mathsf{let}\,x_n \Leftarrow [\mathsf{term}_{\tau_n}\,\phi_n]\,\mathsf{in}\,e$$

It is routine to verify that this satisfies the required property. □

4.8 Full abstraction

We can now fit together the results from Sections 4.6 and 4.7 to show that the semantics for $\mathsf{NMML}\Sigma$ is fully abstract. In Section 4.6 we showed that the denotational characterization and proof system for the program logic were equivalent:

$$\Delta \vdash e : \phi \text{ iff } [\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]$$

In this section we can extend this to show (as long as the semantics for Σ is adequate and expressive) that:

$$\Delta \vdash e : \phi \text{ implies } \Delta \models e : \phi \text{ implies } [\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]$$

and so the operational characterization of the program logic is equivalent to the denotational characterization and to the proof system. From this we can prove full abstraction.

The proof of full abstraction relies on the expressivity of $\mathsf{NMML}\Sigma$, and thus on the expressivity of Σ. If we have a semantics for Σ that is adequate but not expressive, then we can still show that the semantics for $\mathsf{NMML}\Sigma$ is correct, although we cannot show that it is fully abstract.

PROPOSITION 21.

1. *If $e \xrightarrow{\iota} e'$ then $[\![\vdash e : \mathsf{C}\tau]\!] \geq [\![\vdash e : \mathsf{C}\tau]\!]$.*
2. *If $e \xrightarrow{\surd} f$ then $[\![\vdash e : \mathsf{C}\tau]\!] \geq [\![\vdash [f] : \mathsf{C}\tau]\!]$.*

PROOF. An induction on the proof of reduction. □

PROPOSITION 22. *If a semantics for Σ is adequate, then $\Delta \vdash e : \phi$ implies $\Delta \models e : \phi$.*

PROOF. First show by induction on the proof of $\phi \leq \psi$ that if $\phi \leq \psi$ and $\models e : \phi$ then $\models e : \psi$. The result then follows by an induction on the proof of $\Delta \vdash e : \phi$. □

THEOREM 23 (CORRECTNESS). *If a semantics for Σ is adequate, then its extension to $\mathsf{NMML}\Sigma$ is correct.*

PROOF. If $\Gamma \vdash e \sqsubseteq_D f : \tau$ then for any closing context $C[\_]$ of type $\mathsf{C}I$ we have:

$$C[e] \stackrel{\surd}{\Longrightarrow} *$$

$$\begin{array}{ll}
\Rightarrow [\![[*]]\!] \leq [\![\vdash C[e] : \mathsf{C}I]\!] & \text{(Propn 21)} \\
\Rightarrow [\![[*]]\!] \leq [\![\vdash C[f] : \mathsf{C}I]\!] & \text{(Hypothesis)} \\
\Rightarrow \vdash C[f] : [*] & \text{(Propn 18)} \\
\Rightarrow \models C[f] : [*] & \text{(Propn 22)} \\
\Rightarrow C[e] \stackrel{\surd}{\Longrightarrow} * & \text{(Defn of } \models\text{)}
\end{array}$$

Thus $\Gamma \vdash e \sqsubseteq_O f : \tau$. □

PROPOSITION 24. *If a semantics for* Σ *is expressive and adequate, then* $\Delta \models e : \phi$ *implies* $[\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]$.

PROOF. Using Proposition 21, show by induction on the proof of $\models e : \phi$ that if $\models e : \phi$ then $[\![\vdash e : \tau]\!]\bot \geq [\![\phi]\!]$.

The only difficult case is for functions, where we have deduced $\models e : \phi \Rightarrow \psi$ from $\forall \models f : \phi \,.\, \models ef : \psi$. By expressivity and Propositions 18, 19 and 22 we have $\models \mathsf{term}_\sigma \phi : \phi$, so by the hypothesis we have $\models e(\mathsf{term}_\sigma \phi) : \psi$, so by induction $[\![\psi]\!] \leq [\![\vdash e(\mathsf{term}_\sigma \phi) : \mathsf{C}\tau]\!]\bot$, from which it is routine to deduce the desired result that $[\![\phi \Rightarrow \psi]\!] \leq [\![\vdash e : \sigma \to \mathsf{C}\tau]\!]$.

Then we have:

$$\begin{array}{ll}
\vec{x} : \vec{\phi} \models e : \phi & \\
\quad \Rightarrow \forall \models \vec{f} : \vec{\phi} \,.\, e[\vec{f}/\vec{x}] : \phi & \text{(Defn of } \models\text{)} \\
\quad \Rightarrow \models e[\mathsf{term}\,\vec{\phi}/\vec{x}] : \phi & (\models \mathsf{term}\,\phi : \phi) \\
\quad \Rightarrow [\![\phi]\!] \leq [\![\vdash e[\mathsf{term}\,\vec{\phi}/\vec{x}] : \tau]\!]\bot & \text{(Above)} \\
\quad \Rightarrow [\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!]([\![\vdash \mathsf{term}\,\vec{\phi} : \vec{\tau}]\!]\bot) & \text{(NMML-algebra)} \\
\quad \Rightarrow [\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\vec{x} : \vec{\phi}]\!] & ([\![\mathsf{term}\,\phi]\!]\bot = [\![\phi]\!])
\end{array}$$

Note that we have used expressivity in this proof, but not in the proof of Proposition 22.□

THEOREM 25 (FULL ABSTRACTION). *If a semantics for* Σ *is expressive and adequate then its extension to* $\mathsf{NMML}\Sigma$ *is fully abstract.*

PROOF. For any $\phi : \mathcal{L}\tau$ and $\Delta : \mathcal{L}\Gamma$ we have:

$$\begin{array}{ll}
[\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!] & \\
\quad \Leftrightarrow [\![[*]]\!] \leq [\![\phi \Rightarrow [*]]\!]([\![\Gamma \vdash e : \tau]\!][\![\Delta]\!]) & \text{(Defn of } \Rightarrow\text{)} \\
\quad \Leftrightarrow [\![[*]]\!] \leq ([\![\Gamma \vdash \mathsf{term}_{\tau \to \mathsf{C}I}(\phi \Rightarrow [*]) : \tau \to \mathsf{C}I]\!][\![\Delta]\!]) & ([\![\Gamma \vdash \mathsf{term}\,\phi]\!][\![\Delta]\!] = [\![\phi]\!]) \\
\qquad\qquad ([\![\Gamma \vdash e : \mathsf{C}I]\!][\![\Delta]\!] & \\
\quad \Leftrightarrow [\![[*]]\!] \leq [\![\Gamma \vdash \mathsf{term}_{\tau \to \mathsf{C}I}(\phi \Rightarrow [*])e : \mathsf{C}I]\!][\![\Delta]\!] & \text{(Application)} \\
\quad \Leftrightarrow [\![[*]]\!] \leq [\![\vdash C_\Delta[\mathsf{term}_{\tau \to \mathsf{C}I}(\phi \Rightarrow [*])e] : \mathsf{C}I]\!]\bot & \text{(Propn 20)} \\
\quad \Leftrightarrow \models C_\Delta[\mathsf{term}_{\tau \to \mathsf{C}I}(\phi \Rightarrow [*])e] : [*] & \text{(Propns 18, 22, 24)} \\
\quad \Leftrightarrow C_\Delta[\mathsf{term}_{\tau \to \mathsf{C}I}(\phi \Rightarrow [*])e] \stackrel{\surd}{\Longrightarrow} * & \text{(Defn of } \models\text{)}
\end{array}$$

Thus if $\Gamma \vdash e \sqsubseteq_O f : \tau$ then:

$$[\![\phi]\!] \leq [\![\Gamma \vdash e : \tau]\!][\![\Delta]\!] \text{ implies } [\![\phi]\!] \leq [\![\Gamma \vdash f : \tau]\!][\![\Delta]\!]$$

Thus since $[\![\tau]\!]$ and $[\![\Gamma]\!]$ are algebraic, and by Proposition 16 every compact element has a corresponding proposition, we have:

$$[\![\Gamma \vdash e : \tau]\!] \leq [\![\Gamma \vdash f : \tau]\!]$$

Thus the semantics for $\mathsf{NMML}\Sigma$ is fully abstract. □

References

8807, I. (1989). *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*.

ABRAMSKY, S. (1989). The lazy lambda calculus. In TURNER, D., editor, *Declarative Programming*. Addison-Wesley.

ABRAMSKY, S. (1991). Domain theory in logical form. *Ann. Pure Appl. Logic*, 51:1–77.

ABRAMSKY, S. and JUNG, A. (1994). Domain theory. To appear in *Handbook of Theoret. Comp. Sci.*

DAVEY, B. A. and PRIESTLEY, H. A. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

GORDON, A. (1995). Bisimilarity as a theory of functional programming. In *Proc. MFPS 95*, number 1 in Electronic Notes in Comp. Sci. Springer-Verlag.

GORDON, A. *et al.*(1994). A proposal for monadic I/O in Haskell 1.3. WWW document, Haskell 1.3 Committee, http://www.cl.cam.ac.uk/users/adg/io.html.

HENNESSY, M. (1988). *Algebraic Theory of Processes*. MIT Press.

HENNESSY, M. (1992). A denotational model for higher-order processes. Technical Report 6/92, University of Sussex.

HENNESSY, M. (1994). A fully abstract denotational model for higher-order processes. *Information and Computation*, 112(1):55–95.

HENNESSY, M. and MILNER, R. (1980). On observing nondeterminism and concurrency. In DE BAKKER, J. W. and VAN LEEUWEN, J., editors, *Proc. ICALP 80*. Springer-Verlag. LNCS 85.

HENNESSY, M. and PLOTKIN, G. (1979). Full abstraction for a simple parallel programming language. In *Proc. MFPS 79*, volume 74 of *Lecture Notes in Computer Science*. Springer Verlag.

HOARE, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

HOWE, D. (1989). Equality in lazy computation systems. In *Proc. LICS '89*, pages 198–203. IEEE Computer Society Press.

HOWE, D. (1992). Proving congruence of simulation orderings in functional languages. Unpublished manuscript.

HUDAK, P., PEYTON JONES, S. L., WADLER, P., *et al.*(1992). A report on the functional language Haskell. *SIGPLAN Notices*.

JEFFREY, A. (1994). A fully abstract semantics for concurrent graph reduction: Extended abstract. In *Proc. LICS '94*. IEEE Computer Society Press.

JEFFREY, A. (1995). A fully abstract semantics for a concurrent functional language with monadic types. In *Proc. LICS 95*, pages 255–264. IEEE Computer Society Press.

JOHNSTONE, P. T. (1982). *Stone Spaces*. Cambridge University Press.

KOZEN, D. (1983). Results on the propositional mu-calculus. *Theoret. Comput. Sci.*, 27:333–354.

LAMBEK, J. and SCOTT, P. J. (1986). *Introduction to Higher Order Categorical Logic*. Cambridge University Press.

LARSEN, K. G., GODSKESEN, J. C., and ZEEBERG, M. (1989). TAV, tools for automatic verifi cation, user manual. Technical report R 89–19, Dept Math. and Comp. Sci., Ålborg University.

MAC LANE, S. (1971). *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag.

MILNER, R. (1989). *Communication and Concurrency*. Prentice-Hall.

MISLOVE, M. and OLES, F. (1992). A simple language supporting angelic nondeterminism and parallel composition. In *Proc. MFPS 91*, volume 598 of *Lecture Notes in Computer Science*. Springer Verlag.

MOGGI, E. (1991). Notions of computation and monad. *Inform. and Comput.*, 93:55–92.

MORRIS, J.-H. (1968). Lambda calculus models of programming languages. Dissertation, M.I.T.

ONG, C.-H. L. (1988). *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. PhD thesis, Imperial College, London University.

ONG, C.-H. L. (1993). Non-determinism in a functional setting. In *Proc. LICS '93*, pages 275–286. IEEE Computer Soc. Press.

PIERCE, B. C. (1991). *Basic Category Theory for Computer Scientists*. MIT press.

PLOTKIN, G. (1977). LCF considered as a programming language. *Theoret. Comput. Sci.*, 5:223–256.

PLOTKIN, G. (1981). Postgraduate lecture notes in advanced domain theory. Including the 'Pisa notes'.

REPPY, J. (1991). A higher-order concurrent langauge. In *Proc. SIGPLAN 91*, pages 294–305.

REPPY, J. (1992). *Higher-Order Concurrency*. Ph.D thesis, Cornell Univ.

SCOTT, D. S. (1982). Domains for denotational semantics. In NEILSEN, M. and SCHMIDT, E. M., editors, *Proc. ICALP 82*, pages 577–613. Springer-Verlag. LNCS 140.

THOMSEN, B. (1989). A calculus of higher order communicating systems. In *Proc. POPL '89*, pages 143–154.

WADLER, P. (1990). Comprehending monads. In *Proc. 1990 ACM Conf. Lisp and Functional Programming*, pages 61–78.