Monads and Effects

Nick Benton[1], John Hughes[2], and Eugenio Moggi[3]*

[1] Microsoft Research
[2] Chalmers Univ.
[3] DISI, Univ. di Genova, Genova, Italy
moggi@disi.unige.it

Abstract. A tension in language design has been between simple semantics on the one hand, and rich possibilities for side-effects, exception handling and so on on the other. The introduction of monads has made a large step towards reconciling these alternatives. First proposed by Moggi as a way of structuring semantic descriptions, they were adopted by Wadler to structure Haskell programs, and now offer a general technique for delimiting the scope of effects, thus reconciling referential transparency and imperative operations within one programming language. Monads have been used to solve long-standing problems such as adding pointers and assignment, inter-language working, and exception handling to Haskell, without compromising its purely functional semantics. The course will introduce monads, effects and related notions, and exemplify their applications in programming (Haskell) and in compilation (MLj). The course will present typed metalanguages for monads and related categorical notions, and describe how they can be further refined by introducing effects.

* Research partially supported by MURST and ESPRIT WG APPSEM.

1 Monads and Computational Types

Monads, sometimes called triples, have been considered in Category Theory (CT) relatively late (only in the late fifties). Monads and comonads (the dual of monads) are closely related to adjunctions, probably the most pervasive notion in CT. The connection between monads and adjunctions was established independently by Kleisli and Eilenberg-Moore in the sixties. Monads, like adjunctions, arise in many contexts (e.g. in algebraic theories). Good CT references for monads are [Man76,BW85,Bor94]. It is not surprising that monads arise also in applications of CT to Computer Science (CS). We will use monads for giving denotational semantics to programming languages, and more specifically as a way of modeling *computational types* [Mog91]:

> ... to interpret a programming language in a category $\mathcal{C}$, we distinguish the object A of values (of type A) from the object TA of computations (of type A), and take as denotations of programs (of type A) the *elements* of TA. In particular, we identify the type A with the object of values (of type A) and obtain the object of computations (of type A) by applying an unary type-constructor T to A. We call T a *notion of computation*, since it abstracts away from the type of values computations may produce.

Example 1. We give few notions of computation in the category of sets.

- **partiality** $TA = A_\perp$, i.e. $A + \{\perp\}$, where $\perp$ is the *diverging computation*
- **nondeterminism** $TA = \mathcal{P}_{fin}(A)$, i.e. the set of finite subsets of A
- **side-effects** $TA = (A \times S)^S$, where S is a set of states, e.g. a set U^L of stores or a set of input/output sequences U^*
- **exceptions** $TA = A + E$, where E is the set of exceptions
- **continuations** $TA = R^{(R^A)}$, where R is the set of results
- **interactive input** $TA = (\mu X.A + X^U)$, where U is the set of characters. More explicitly TA is the set of U-branching trees with only finite paths and A-labelled leaves
- **interactive output** $TA = (\mu X.A + (U \times X))$, i.e. $U^* \times A$ up to iso.

Further examples (in the category of cpos) could be given based on the denotational semantics for various programming languages

Remark 2. Many of the examples above are instances of the following one: given a single sorted algebraic theory Th, let $TA = |T_{Th}(A)|$, i.e. the carrier of the free Th-algebra $T_{Th}(A)$ over A. One could consider *combinations* of the examples above, e.g.

- $TA = ((A + E) \times S)^S$ and $TA = ((A \times S) + E)^S$ capture imperative programs with exceptions
- $TA = \mu X.\mathcal{P}_{fin}(A + (Act \times X))$ captures parallel programs interacting via a set Act of actions (in fact TA is the set of finite synchronization trees up to strong bisimulation)

– $TA = \mu X.\mathcal{P}_{fin}((A + X)\times S)^S$ captures parallel imperative programs with shared memory.

[Wad92] advocates a similar idea to mimic impure programs in a pure functional language. Indeed the Haskell community has gone a long way in exploiting this approach to reconcile the advantages of pure functional programming with the flexibility of imperative (or other styles of) programming. The analogies of computational types with *effect systems* [GL86] have been observed by [Wad92], but formal relations between the two have been established only recently (e.g. see [Wad98]).

In the denotational semantics of programming languages there are other informal notions modeled using monads, for instance *collection types* in database languages [BNTW95] or *collection classes* in object-oriented languages [Man98]. It is important to distinguish the mathematical notion of monad (or its refinements) from informal notions, such as computational and collection types, which are *defined* by examples. In fact, these informal notions can be modeled with a better degree of approximation by considering monads with additional properties or additional structures. When considering these refinements, it is often the case that what seems a natural requirement for modeling computational types is not appropriate for modeling collection types, for instance:

– It seems natural that programming languages can express divergent computations and more generally they should support recursive definitions of programs; therefore computational types should come equipped with a constant $\bot : TA$ for the divergent computation and a (least) fix-point combinator $Y : (TA \to TA) \to TA$.
– While it seems natural that a collection should have only a finite number of elements, and there should be an empty collection $\emptyset : TA$ and a way of merging two collections using a binary operation $+ : TA \to TA \to TA$.

There are several equivalent definitions of monad (the same happens with adjunctions). [Man76] gives three definitions of monad/triple called: in monoid form (the one usually adopted in CT books), in extension form (the most intuitive one for us), and in clone form (which takes composition in the Kleisli category as basic). We consider only triples in monoid and extension form.

Notation 1 *We assume knowledge of basic notions from category theory, such as category, functor and natural transformation. In some cases familiarity with universal constructions (products, sums, exponentials) and adjunction is assumed. We use the following notation:*

– *given a category* $\mathcal{C}$ *we write:*
  $|\mathcal{C}|$ *for the set/class of its objects,*
  $\mathcal{C}(A, B)$ *for the hom-set of morphisms from* A *to* B*,*
  $g \circ f$ *and* $f; g$ *for the composition* $A \xrightarrow{f} B \xrightarrow{g} C$*,*
  id_A *for the identity on* A
– $F : \mathcal{C} \to \mathcal{D}$ *means that* F *is a functor from* $\mathcal{C}$ *to* $\mathcal{D}$*, and*
  $\sigma : F \dot{\to} G$ *means that* σ *is a natural transformation from* F *to* G

– $\mathcal{C} \underset{F}{\overset{G}{\underset{\longrightarrow}{\overset{\longleftarrow}{\top}}}} \mathcal{D}$ *means that* G *is right adjoint to* F *(*F *is left adjoint to* G*).*

Definition 3 (Kleisli triple/triple in extension form). *A* **Kleisli triple** *over a category* $\mathcal{C}$ *is a triple* $(T, \eta, _^*)$*, where* $T : |\mathcal{C}| \to |\mathcal{C}|$*,* $\eta_A : A \to TA$ *for* $A \in |\mathcal{C}|$*,* $f^* : TA \to TB$ *for* $f : A \to TB$ *and the following equations hold:*

– $\eta_A^* = \mathrm{id}_{TA}$
– $\eta_A ; f^* = f$ *for* $f : A \to TB$
– $f^* ; g^* = (f ; g^*)^*$ *for* $f : A \to TB$ *and* $g : B \to TC$*.*

Kleisli triples have an intuitive justification in terms of computational types

– η_A is the *inclusion* of values into computations

$$a : A \overset{\eta_A}{\longmapsto} [a] : TA$$

– f^* is the *extension* of a function f from values to computations to a function from computations to computations, which first evaluates a computation and then applies f to the resulting value

$$\frac{a : A \overset{f}{\longmapsto} f\ a : TB}{c : TA \overset{f^*}{\longmapsto} \mathrm{let}\ a \Leftarrow c\ \mathrm{in}\ f\ a : TB}$$

In order to justify the axioms for a Kleisli triple we have first to introduce a category $\mathcal{C}_T$ whose morphisms correspond to programs. We proceed by analogy with the categorical semantics for terms, where types are interpreted by objects and terms of type B with a parameter (free variable) of type A are interpreted by morphisms from A to B. Since the denotation of programs of type B are supposed to be elements of TB, programs of type B with a parameter of type A ought to be interpreted by morphisms with codomain TB, but for their domain there are two alternatives, either A or TA, depending on whether parameters of type A are identified with values or computations of type A. We choose the first alternative, because it entails the second. Indeed computations of type A are the same as values of type TA. So we take $\mathcal{C}_T(A, B)$ to be $\mathcal{C}(A, TB)$. It remains to define composition and identities in $\mathcal{C}_T$ (and show that they satisfy the unit and associativity axioms for categories).

Definition 4 (Kleisli category). *Given a Kleisli triple* $(T, \eta, _^*)$ *over* $\mathcal{C}$*, the* **Kleisli category** $\mathcal{C}_T$ *is defined as follows:*

– *the objects of* $\mathcal{C}_T$ *are those of* $\mathcal{C}$
– *the set* $\mathcal{C}_T(A, B)$ *of morphisms from* A *to* B *in* $\mathcal{C}_T$ *is* $\mathcal{C}(A, TB)$
– *the identity on* A *in* $\mathcal{C}_T$ *is* $\eta_A : A \to TA$
– $f \in \mathcal{C}_T(A, B)$ *followed by* $g \in \mathcal{C}_T(B, C)$ *in* $\mathcal{C}_T$ *is* $f ; g^* : A \to TC$*.*

It is natural to take η_A as the identity on A in the category $\mathcal{C}_T$, since it maps a parameter x to $[x]$, i.e. to x viewed as a computation. Similarly composition in $\mathcal{C}_T$ has a simple explanation in terms of the intuitive meaning of f^*, in fact

$$\frac{x : A \stackrel{f}{\longmapsto} f\ x : TB \quad y : B \stackrel{g}{\longmapsto} g\ y : TC}{x : A \stackrel{f;g^*}{\longmapsto} \text{let } y \Leftarrow f\ x \text{ in } g\ y : TC}$$

i.e. f followed by g in $\mathcal{C}_T$ with parameter x is the program which first evaluates the program $f\ x$ and then feed the resulting value as parameter to g. At this point we can give also a simple justification for the three axioms of Kleisli triples, namely they are equivalent to the following unit and associativity axioms, which say that $\mathcal{C}_T$ is a category:

- $f; \eta_B^* = f$ for $f : A \to TB$
- $\eta_A; f^* = f$ for $f : A \to TB$
- $(f; g^*); h^* = f; (g; h^*)^*$ for $f : A \to TB$, $g : B \to TC$ and $h : C \to TD$.

Example 5. We go through the examples of computational types given in Example 1 and show that they are indeed part of suitable Kleisli triples.

- **partiality** $TA = A_\perp (= A + \{\perp\})$
  η_A is the inclusion of A into $A_\perp$
  if $f : A \to TB$, then $f^*\ \perp = \perp$ and $f^*\ a = f\ a$ (when $a \in A$)
- **nondeterminism** $TA = \mathcal{P}_{fin}(A)$
  η_A is the singleton map $a \mapsto \{a\}$
  if $f : A \to TB$ and $c \in TA$, then $f^*\ c = \cup\{f\ x | x \in c\}$
- **side-effects** $TA = (A \times S)^S$
  η_A is the map $a \mapsto \lambda s : S.(a, s)$
  if $f : A \to TB$ and $c \in TA$, then $f^*\ c = \lambda s : S.(\text{let } (a, s') = c\ s \text{ in } f\ a\ s')$
- **exceptions** $TA = A + E$
  η_A is the injection map $a \mapsto \text{inl } a$
  if $f : A \to TB$, then $f^*(\text{inr } e) = \text{inr } e$ (where $e \in E$) and $f^*(\text{inl } a) = f\ a$ (where $a \in A$)
- **continuations** $TA = R^{(R^A)}$
  η_A is the map $a \mapsto (\lambda k : R^A.k\ a)$
  if $f : A \to TB$ and $c \in TA$, then $f^*\ c = (\lambda k : R^B.c(\lambda a : A.f\ a\ k))$
- **interactive input** $TA = (\mu X.A + X^U)$
  η_A maps a to the tree consisting only of one leaf labelled with a
  if $f : A \to TB$ and $c \in TA$, then $f^*\ c$ is the tree obtained by replacing leaves of c labelled by a with the tree $f\ a$
- **interactive output** $TA = (\mu X.A + (U \times X))$
  η_A is the map $a \mapsto (\epsilon, a)$
  if $f : A \to TB$, then $f^*\ (s, a) = (s * s', b)$, where $f\ a = (s', b)$ and $s * s'$ is the concatenation of s followed by s'.

Exercise 6. Define Kleisli triples in the category of cpos similar to those given in Example 5, but ensure that each computational type TA has a least element $\perp$. DIFFICULT: in cpos there are three Kleisli triple for nondeterminism, one for each powerdomain construction.

Exercise 7. When modeling a programming language the first choice to make is which category to use. For instance, it is impossible to find a monad over the category of sets which support recursive definitions of programs, one should work in the category of cpos (or similar categories). Moreover, there are other aspects of programming languages that are orthogonal to computational types, e.g. recursive and polymorphic types, that cannot be models in the category of sets (one has to work in categories like that of cpos or in *realizability models*).

If one wants to model a two-level language, where there is a notion of static and dynamic, then the following categories are particularly appropriate

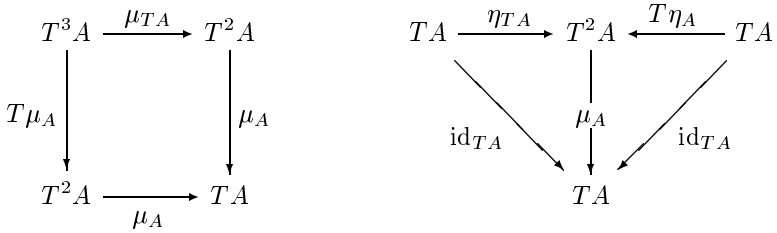
- the category $s(\mathcal{C})$, where $\mathcal{C}$ can be any CCC, is defined as follows
  an object is a pair (A_s, A_d) with $A_s, A_d \in |\mathcal{C}|$, A_s is the static and A_d is the dynamic part;
  a morphism in $s(\mathcal{C})((A_s, A_d), (B_s, B_d))$ is a pair (f_s, f_d) with $f_s \in \mathcal{C}(A_s, B_s)$ and $f_d \in \mathcal{C}(A_s \times A_d, B_d)$, thus the static part of the result depends only on the static part of the input.
- the category $Fam(\mathcal{C})$, where $\mathcal{C}$ can be any CCC with small limits, is defined as follows
  an object is a family $(A_i | i \in I)$ with I a set and $A_i \in |\mathcal{C}|$ for every $i \in I$;
  a morphism in $Fam(\mathcal{C})((A_i | i \in I), (B_j | j \in J))$ is a pair (f, g) with $f : I \to J$ and g is an I-index family of morphisms s.t. $g_i \in \mathcal{C}(A_i, B_{fi})$ for every $i \in I$.

Define Kleisli triples in the categories $s(\mathcal{C})$ and $Fam(\mathcal{C})$ similar to those given in Example 5 (assume that $\mathcal{C}$ is the category of sets). Notice that in a two-level language static and dynamic computations don't have to be the same.

1.1 Monads and Related Notions

This section contains definitions and facts, that are not essential to the subsequent developments. First we establish the equivalence of Kleisli triples and monads.

Definition 8 (Monad/triple in monoid form). *A* **monad** *over $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : \mathrm{id}_{\mathcal{C}} \dot{\to} T$ and $\mu : T^2 \dot{\to} T$ are natural transformations and the following diagrams commute:*

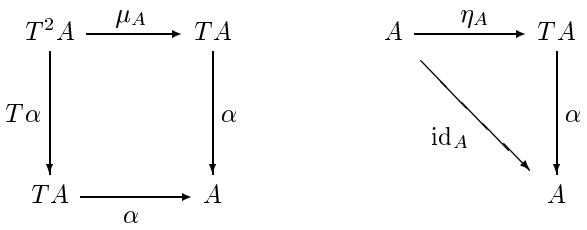


Proposition 9. *There is a bijection between Kleisli triples and monads.*

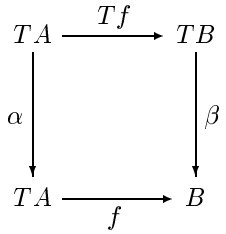
Proof. Given a Kleisli triple $(T, \eta, _^*)$, the corresponding monad is (T, η, μ), where T is the extension of the function T to an endofunctor by taking $T\ f = (f; \eta_B)^*$ for $f : A \to B$ and $\mu_A = \mathrm{id}^*_{TA}$. Conversely, given a monad (T, η, μ), the corresponding Kleisli triple is $(T, \eta, _^*)$, where T is the restriction of the functor T to objects and $f^* = (T\ f); \mu_B$ for $f : A \to TB$.

Definition 10 (Eilenberg-Moore category). *Given a monad (T, η, μ) over $\mathcal{C}$, the* **Eilenberg-Moore category** $\mathcal{C}^T$ *is defined as follows:*

- *the objects of $\mathcal{C}^T$ are T-***algebras***, i.e. morphisms $\alpha : TA \to A$ in $\mathcal{C}$ s.t.*



  A is called the **carrier** *of the T-algebra α*
- *a morphism $f \in \mathcal{C}^T(\alpha, \beta)$ from $\alpha : TA \to A$ to $\beta : TB \to B$ is a morphism $f : A \to B$ in $\mathcal{C}$ s.t.*



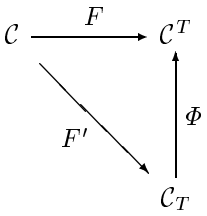
  identity and composition in $\mathcal{C}^T$ are like in $\mathcal{C}$.

Any adjunction $\mathcal{C} \underset{F}{\overset{G}{\underset{\longrightarrow}{\overset{\longleftarrow}{\top}}}} \mathcal{D}$ induces a monad over $\mathcal{C}$ with $T = F; G$. The Kleisli and Eilenberg-Moore categories can be used to prove the converse, i.e. that any monad over $\mathcal{C}$ is induced by an adjunction. Moreover, the Kleisli category can be identified with the full sub-category of $\mathcal{C}^T$ consisting of the *free T-algebras*.

Proposition 11. *Given a monad (T, η, μ) over $\mathcal{C}$ there are two adjunctions*

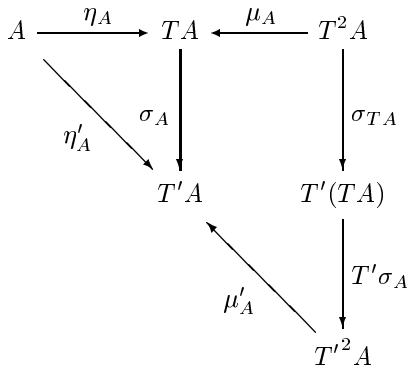
$$\mathcal{C} \underset{F}{\overset{U}{\underset{\longrightarrow}{\overset{\longleftarrow}{\top}}}} \mathcal{C}^T \qquad\qquad \mathcal{C} \underset{F'}{\overset{U'}{\underset{\longrightarrow}{\overset{\longleftarrow}{\top}}}} \mathcal{C}_T$$

which induce T. Moreover, there is a full and faithful functor $\Phi : \mathcal{C}_T \to \mathcal{C}^T$ s.t.



Proof. The action of functors on objects are as follows: $U(\alpha : TA \to A) \stackrel{\Delta}{=} A$, $FA \stackrel{\Delta}{=} \mu_A : T^2A \to TA$, $U'A \stackrel{\Delta}{=} TA$, $F'A \stackrel{\Delta}{=} A$, and $\Phi A \stackrel{\Delta}{=} \mu_A : T^2A \to TA$.

Definition 12 (Monad morphism). *Given (T, η, μ) and (T', η', μ') monads over $\mathcal{C}$, a* **monad-morphism** *from the first to the second is a natural transformation $\sigma : T \dot{\to} T'$ s.t. :*



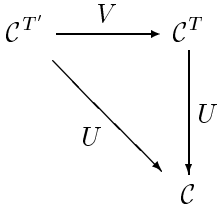
An equivalent definition of monad morphism (in terms of Kleisli triples) is a family of morphisms $\sigma_A : TA \to T'A$ for $A \in |\mathcal{C}|$ s.t.

- $\eta_A ; \sigma_A = \eta'_A$
- $f^* ; \sigma_B = \sigma_A ; (f ; \sigma_B)^{*'}$ *for* $f : A \to TB$

We write $Mon(\mathcal{C})$ for the **category of monads over** $\mathcal{C}$ *and monad morphisms.*

There is also a more general notion of monad morphism, which does not require that the monads are over the same category. Monad morphisms allow to view T'-algebras as T-algebras with the same underlying *carrier*, more precisely

Proposition 13. *There is a bijective correspondence between monad morphisms $\sigma : T \to T'$ and functors $V : \mathcal{C}^{T'} \to \mathcal{C}^T$ s.t.*



Proof. The action of V on objects is $V(\alpha' : T'A \to A) \stackrel{\Delta}{=} \sigma_A; \alpha' : TA \to A$, and σ_A is defined in terms of V as $\sigma_A \stackrel{\Delta}{=} TA \xrightarrow{T\eta'_A} T(T'A) \xrightarrow{V\ \mu'_A} T'A$.

Remark 14. [Fil99] uses a *layering* $\zeta_A : T(T'A) \to T'A$ of T' over T in place of a monad morphism $\sigma_A : TA \to T'A$. The two notions are equivalent, in particular ζ_A is given by $V\ \mu'_A$, i.e. $\sigma_{T'A}; \mu'_A$.

2 Metalanguages with Computational Types

It is quite inconvenient to work directly in a specific category or with a specific monad. Mathematical logic provides a simple solution to abstract away from specific models: fix a language, define what is an interpretation of the language in a model, find a formal system (on the language) that capture the desired properties of models. When the formal system is sound, one can forget about the models and use the formal system instead. Moreover, if the formal system is also complete, then nothing is lost (as far as one is concerned with properties expressible in the language, and valid in all models). Several formal systems have been proved sound and complete w.r.t. certain class of categories:

- many sorted equational logic corresponds to categories with finite products;
- simply typed λ-calculus corresponds to cartesian closed categories (CCC);
- intuitionistic higher-order logic corresponds to elementary toposes.

Remark 15. To ensure soundness w.r.t. the given classes of models, the above formal systems should cope with the possibility of empty carriers. While in mathematical logic it is often assume that all carriers are inhabited. Categorical Logic is the branch of CT devoted mainly at establishing links between formal systems and classes of categorical structures.

Rather than giving a complete formal system, we say how to add computational types to your favorite formal system (for instance higher-order λ-calculus, or a λ-calculus with dependent types like a logical framework). The only assumption we make is that the formal system should include many sorted equational logic (this rules out systems like the *linear λ-calculus*). More specifically we assume that the formal system has the following judgments

- $\Gamma \vdash$, i.e. Γ is a well-formed context
- $\Gamma \vdash \tau\ type$, i.e. τ is a well-formed type in context Γ
- $\Gamma \vdash e : \tau$, i.e. e is a well-formed term of type τ in context Γ
- $\Gamma \vdash \phi\ prop$, i.e. ϕ is a well-formed proposition in context Γ
- $\Gamma \vdash \phi$, i.e. the well-formed proposition ϕ in context Γ is true

and that the following rules are part of the formal system (or derivable)

- $\dfrac{}{\emptyset \vdash}$ $\qquad \dfrac{\Gamma \vdash \tau\ type}{\Gamma, x : \tau \vdash}\ x \notin \mathrm{DV}(\Gamma)$ $\qquad \dfrac{\Gamma \vdash}{\Gamma \vdash x : \tau}\ \tau = \Gamma(x)$

– $\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 = e_2 : \tau)\ \mathit{prop}}$ this says when an equation is well-formed

– weak $\dfrac{\Gamma \vdash \tau \quad \Gamma \vdash \phi}{\Gamma, x : \tau \vdash \phi}\ x \notin \mathrm{DV}(\Gamma)$ sub $\dfrac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash \phi}{\Gamma \vdash \phi[x := e]}$

– $\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash e = e : \tau}$ $\dfrac{\Gamma \vdash e_1 = e_2 : \tau}{\Gamma \vdash e_2 = e_1 : \tau}$ $\dfrac{\Gamma \vdash e_1 = e_2 : \tau \quad \Gamma \vdash e_2 = e_3 : \tau}{\Gamma \vdash e_1 = e_3 : \tau}$

cong $\dfrac{\Gamma, x : \tau \vdash \phi\ \mathit{prop} \quad \Gamma \vdash e_1 = e_2 : \tau \quad \Gamma \vdash \phi[x := e_1]}{\Gamma \vdash \phi[x := e_2]}$

Remark 16. More complex formal systems may require other forms of judgment, e.g. equality of types (and contexts), or other *sorts* besides *type* (along the line of Pure Type Systems). The categorical interpretation of typed calculi, including those with dependent types, is described in [Pit00b,Jac99].

The rules for adding computational types are

– $T\ \dfrac{\Gamma \vdash \tau\ \mathit{type}}{\Gamma \vdash T\tau\ \mathit{type}}$ lift $\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash [e]_T : T\tau}$

let $\dfrac{\Gamma \vdash e_1 : T\tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : T\tau_2}{\Gamma \vdash \mathrm{let}_T\, x \Leftarrow e_1\, \mathrm{in}\, e_2 : T\tau_2}\ x \notin \mathrm{FV}(\tau_2)$

$[e]_T$ is the program/computation that simply returns the value e, while $\mathrm{let}_T\, x \Leftarrow e_1\, \mathrm{in}\, e_2$ is the computation which first evaluates e_1 and binds the result to x, then evaluates e_2.

In calculi without dependent types the side-condition $x \notin \mathrm{FV}(\tau_2)$ in the let-rule is automatically satisfied. From now on we ignore such side-conditions.

– let.ξ $\dfrac{\Gamma \vdash e : T\tau_1 \quad \Gamma, x : \tau_1 \vdash e_1 = e_2 : T\tau_2}{\Gamma \vdash \mathrm{let}_T\, x \Leftarrow e\, \mathrm{in}\, e_1 = \mathrm{let}_T\, x \Leftarrow e\, \mathrm{in}\, e_2 : T\tau_2}$

this rule expresses congruence for the let-binder.

– assoc $\dfrac{\Gamma \vdash e_1 : T\tau_1 \quad \Gamma, x_1 : \tau_1 \vdash e_2 : T\tau_2 \quad \Gamma, x_2 : \tau_2 \vdash e_3 : T\tau_3}{\Gamma \vdash \mathrm{let}_T\, x_2 \Leftarrow (\mathrm{let}_T\, x_1 \Leftarrow e_1\, \mathrm{in}\, e_2)\, \mathrm{in}\, e_3 = \mathrm{let}_T\, x_1 \Leftarrow e_1\, \mathrm{in}\, (\mathrm{let}_T\, x_2 \Leftarrow e_2\, \mathrm{in}\, e_3) : T\tau_3}$

this rule says that only the order of evaluation matters (not the parentheses).

– T.β $\dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : T\tau_2}{\Gamma \vdash \mathrm{let}_T\, x \Leftarrow [e_1]_T\, \mathrm{in}\, e_2 = e_2[x := e_1] : T\tau_2}$

T.η $\dfrac{\Gamma \vdash e : T\tau}{\Gamma \vdash \mathrm{let}_T\, x \Leftarrow e\, \mathrm{in}\, [x]_T = e : T\tau}$

these rules say how to eliminate trivial computations (i.e. of the form $[e]_T$).

Remark 17. [Mog91] describes the interpretation of computational types in a simply typed calculus, and establishes soundness and completeness results, while [Mog95] extends such results to logical systems including *evaluation modalities* proposed by Pitts.

For interpreting computational types monads are not enough, one has to use *parameterized* monads. The parameterization is directly related to the form of type-dependency allowed by the typed calculus under consideration. The need to

consider parametrized forms of categorical notions is by now a well-understood fact in categorical logic (it is not a peculiarity of computational types).

We sketch the categorical interpretation in a category $\mathcal{C}$ with finite products of a simply typed metalanguage with computational types (see [Mog91] for more details). The general pattern for interpreting a simply typed calculus according to Lawvere's functorial semantics goes as follows

- a context $\Gamma \vdash$ and a type $\vdash \tau\ type$ are interpreted by objects of $\mathcal{C}$, by abuse of notation we indicate these objects with Γ and τ respectively;
- a term $\Gamma \vdash e : \tau$ is interpreted by a morphism $f : \Gamma \to \tau$ in $\mathcal{C}$;
- a (well formed) equational $\Gamma \vdash e_1 = e_2 : \tau$ is true iff $f_1 = f_2 : \Gamma \to \tau$ as morphisms in $\mathcal{C}$.

Figure 1 gives the relevant clauses of the interpretation. Notice that for interpreting let one needs a parameterized extension operation $_^*$, which maps $f : C \times A \to TB$ to $f^* : C \times TA \to TB$.

RULE	SYNTAX	SEMANTICS
T	$\vdash \tau\ type$	$= \tau$
	$\vdash T\tau\ type$	$= T\tau$
lift	$\Gamma \vdash e : \tau$	$= f : \Gamma \to \tau$
	$\Gamma \vdash [e]_T : T\tau$	$= f; \eta_\tau : \Gamma \to T\tau$
let	$\Gamma \vdash e_1 : T\tau_1$	$= f_1 : \Gamma \to T\tau_1$
	$\Gamma, x : \tau_1 \vdash e_2 : T\tau_2$	$= f_2 : \Gamma \times \tau_1 \to T\tau_2$
	$\Gamma \vdash \mathrm{let}_T\, x \Leftarrow e_1\, \mathrm{in}\, e_2 : T\tau_2$	$= (\mathrm{id}_\Gamma, f_1); f_2^* : \Gamma \to T\tau_2$

Fig. 1. simple interpretation of computational types

2.1 Syntactic Sugar and Alternative Presentations

One can define convenient derived notation, for instance:

- an iterated-let $(\mathrm{let}_T\, \overline{x} \Leftarrow \overline{e}\, \mathrm{in}\, e)$, which is defined by induction on $|\overline{e}| = |\overline{x}|$

  $$\mathrm{let}_T\, \emptyset \Leftarrow \emptyset\, \mathrm{in}\, e \stackrel{\Delta}{\equiv} e \qquad \mathrm{let}_T\, x_0, \overline{x} \Leftarrow e_0, \overline{e}\, \mathrm{in}\, e \stackrel{\Delta}{\equiv} \mathrm{let}_T\, x_0 \Leftarrow e_0\, \mathrm{in}\, (\mathrm{let}_T\, \overline{x} \Leftarrow \overline{e}\, \mathrm{in}\, e)$$

  Haskell's do-notation, inspired by monad comprehension (see [Wad92]), extends the iterated-let by allowing pattern matching and local definitions

In higher-order λ-calculus, the type- and term-constructors can be replaced by constants:

- T becomes a constant of kind $\bullet \to \bullet$, where $\bullet$ is the kind of all types;
- $[e]_T$ and $\mathrm{let}_T\, x \Leftarrow e_1 \,\mathrm{in}\, e_2$ are replaced by polymorphic constants

$$unit_T : \forall X : \bullet . X \to TX \qquad let_T : \forall X, Y : \bullet . (X \to TY) \to TX \to TY$$

where $unit_T \stackrel{\Delta}{=} \Lambda X : \bullet . \lambda x : X . [x]_T$ and
$let_T \stackrel{\Delta}{=} \Lambda X, Y : \bullet . \lambda f : X \to TY . \lambda c : TX . \mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, f\ x$.

In this way the rule (let.ξ) follows from the ξ-rule for λ-abstraction, and the other three equational rules can be replaced with three equational axioms without premises, e.g. $T.\beta$ can be replaced by

$$X, Y : \bullet, x : X, f : X \to TY \vdash \mathrm{let}_T\, x \Leftarrow [x]_T \,\mathrm{in}\, f\ x = f\ x : TY$$

The polymorphic constant $unit_T$ corresponds to the natural transformation η. In higher-order λ-calculus one can define also polymorphic constants

$$map_T : \forall X, Y : \bullet . (X \to Y) \to TX \to TY \qquad flat_T : \forall X : \bullet . T^2 X \to TX$$

corresponding to the action of the functor T on morphisms and to the natural transformation μ

- $map_T \stackrel{\Delta}{=} \Lambda X, Y : \bullet . \lambda f : X \to Y . \lambda c : TX . \mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, [f\ x]_T$
- $flat_T \stackrel{\Delta}{=} \Lambda X : \bullet . \lambda c : T^2 X . \mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, x$

The axiomatization taking as primitive the polymorphic constants $unit_T$ and let_T amounts to the definition of triple in extension form, one can envisage an alternative axiomatization corresponding to that of triple in monoid form, which takes as primitive the polymorphic constants map_T, $unit_T$ and $flat_T$.

2.2 Categorical Definitions in the Metalanguage

The main point for introducing a metalanguage is to provide an alternative to working directly with models/categories. In particular, one expect that categorical notions related to monads, such as algebra and monad morphisms, can be reformulated axiomatically in a metalanguage with computational types.

Definition 18 (Eilenberg-Moore algebras). *$\alpha : TA \to A$ is a T-algebra iff*

- $x : A \vdash \alpha\ [x]_T = x : A$
- $c : T^2 A \vdash \alpha(\mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, x) = \alpha(\mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, [\alpha\ x]_T) : A$

$f : A \to B$ is a T-algebra morphism from $\alpha : TA \to A$ to $\beta : TB \to B$ iff

- $c : TA \vdash f(\alpha\ c) = \beta(\mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, [fx]_T) : B$

We can consider metalanguages with many computational types, corresponding to different monads on the same category. In particular, to define monad morphisms we use a metalanguage with two computational types T and T'.

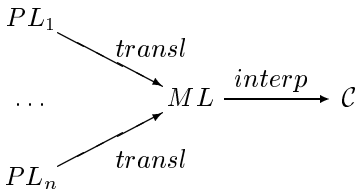
Definition 19 (Monad morphism). *A constant $\sigma : \forall X : \bullet.TX \to T'X$ is a monad morphism from T to T' iff*

- $X : \bullet, x : X \vdash \sigma\ X\ [x]_T = [x]_{T'} : T'X$
- $X, Y : \bullet, c : TX, f : X \to TY \vdash \sigma\ Y\ (\mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, f\ x) = \mathrm{let}_{T'}\, x \Leftarrow \sigma\ X\ c \,\mathrm{in}\, \sigma\ Y\ (f\ x) : T'Y$

3 Metalanguages for Denotational Semantics

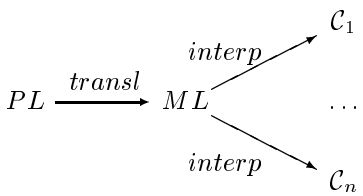
Translation of a language into another provides a simple and general way to give semantics to the first language in terms of a semantics for the second. In denotational semantics it is quite common to define the semantics of a programming language PL by translating it into a typed metalanguage ML. The idea is as old as denotational semantics (see [Sco93]), so the main issue is whether it can be made into a viable technique capable of dealing with complex programming languages. Before being more specific about what metalanguages to use, let us discuss what are the main advantages of semantics via translation:

- to reuse the same ML for translating several programming languages.



  Here we are implicitly assuming that defining a translation from PL to ML is simpler than directly defining an interpretation of PL.
  In this case it is worth putting some effort in the study of ML. In fact, once certain properties of ML have been established (e.g. reasoning principles or computational adequacy), it is usually *easy* to transfer them to PL via the translation.
- to choose ML according to certain criteria, usually not met by programming languages, e.g.
  - a metalanguage built around few *orthogonal* concepts is simpler to study, on the contrary programming languages often introduce *syntactic sugar* for the benefit of programmers;
  - ML may be equipped with a logic so that it can be used for formalizing reasoning principles or for translating specification languages;
  - ML may be chosen as the *internal language* for a class of categories (e.g. CCC) or for a specific semantic category (e.g. that of sets or cpos).

– to use ML for hiding details of semantic categories (see [Gor79]). For instance, when ML is the internal language for a class of categories, it has one intended interpretation in each of them, therefore a translation into ML will induce a variety of interpretations



even when ML has only one intended interpretation, it may be difficult to work with the semantic category directly.

A good starting point for a metalanguage is to build it on top of a fairly standard typed λ-calculus, more controversial issues are:

– whether the metalanguage should be equipped with some logic (ranging from equational logic to higher-order predicate logic).
– whether the metalanguage should be itself a programming language (i.e. to have an operational semantics).

We will discuss how the metalanguages with computational types can help in structuring the translation from PL to ML by the introduction of **auxiliary notation** (see [Mos92,Mog91])

$$PL \xrightarrow{transl} ML(\Sigma) \xrightarrow{transl} ML$$

and by **incrementally defining auxiliary notation** (as advocated in [Fil99], [LHJ95,LH96] and [CM93,Mog97])

$$PL \xrightarrow{transl} ML(\Sigma_n) \xrightarrow{transl} \ldots \xrightarrow{transl} ML(\Sigma_0) \xrightarrow{transl} ML$$

Remark 20. The solutions proposed are closely related to general techniques in *algebraic specifications*, such as abstract datatype, stepwise refinement and hierarchical specifications.

3.1 Computational Types and Structuring

A typical problem of denotational and operational semantics is the following: when a programming language is extended, its semantics may need to be extensively redefined. For instance, when extending a pure functional language with side-effects or exceptions we have to redefine the operational/denotational semantics every time we considered a different extension. The problem remains even when the semantics is given via translation in a typed lambda-calculus: one would keep redefining the translation. In [Mos90] this problem is identified

very clearly, and it is stressed how the use of **auxiliary notation** may help in making semantic definitions more reusable.

[Mog91] identifies *monads* as an important structuring device for denotational semantics (but not for operational semantics!). The basic idea is that there is a unary type constructor T, called a **notion of computation**, and terms of type $T\tau$, should be thought as programs which computes values of type τ. The interpretation of T is not fixed, it varies according to the *computational features* of the programming language under consideration. Nevertheless, one can identifies some operations (for specifying the order of evaluation) and basic properties of them, which should be common to all notions of computation. This suggests to translate a programming language PL into a metalanguage $ML_T(\Sigma)$ with computational types, where the signature Σ gives additional operations (and their properties). In summary, the **monadic approach** to denotational semantics consists of three steps, i.e. given a programming language PL:

- identify a suitable metalanguage $ML_T(\Sigma)$, this hides the interpretation of T and Σ like an interface hides the implementation of an abstract datatype,
- define a translation of PL into $ML_T(\Sigma)$,
- construct a model of $ML_T(\Sigma)$, e.g. via translation into a metalanguage ML without computational types.

By a suitable choice of Σ, one can find a simple translation from PL to $ML_T(\Sigma)$, which usually does not have to be redefined (only extended) when PL is extended, At the same time one can keep the translation of $ML_T(\Sigma)$ into ML fairly manageable.

To exemplify the use of computational types, we consider several programming languages (viewed as λ-calculi with constants), and for each of them we define translations into a metalanguage $ML_T(\Sigma)$ with computational types, for a suitable choice of Σ, and indicate a possible interpretation for computational types and Σ.

CBN Translation: Haskell We consider a simple fragment of Haskell corresponding to the following typed λ-calculus (we ignore issues of type inference, thus terms have explicit type information):

$$
\begin{array}{rll}
\tau \in Type_{Haskell} ::= & \mathsf{Int} & \text{type of integers} \\
& \tau_1 \to \tau_2 & \text{functional type} \\
& \tau_1 \times \tau_2 & \text{product type} \\
e \in Exp_{Haskell} ::= & x & \text{variable} \\
& \mathsf{n} \mid e_0 + e_1 & \text{numerals and integer addition} \\
& \mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \text{conditional} \\
& (\mathrm{let}\, x : \tau = e_1 \,\mathrm{in}\, e_2) & \text{local definition} \\
& \mu x : \tau . e & \text{recursive definition} \\
& \lambda x : \tau . e & \text{abstraction}
\end{array}
$$

$e_1\ e_2$ application

(e_1, e_2) pairing

$\pi_i\ e$ projection

The type system for Haskell derives judgments of the form $\Gamma \vdash e : \tau$ saying

$\tau \in Type_{Haskell}$	$\tau^n \in Type$
Int	Int
$\tau_1 \to \tau_2$	$T\tau_1^n \to T\tau_2^n$
$\tau_1 \times \tau_2$	$T\tau_1^n \times T\tau_2^n$

$e \in Exp_{Haskell}$	$e^n \in Exp$
x	x
n	$[\mathsf{n}]_T$
$e_0 + e_1$	$\mathrm{let}_T\, x_0, x_1 \Leftarrow e_0^n, e_1^n\, \mathrm{in} [x_0 + x_1]_T$
if0 e_0 then e_1 else e_2	$\mathrm{let}_T\, x \Leftarrow e_0^n\, \mathrm{in}\, \mathrm{if}\ x = 0\ \mathrm{then}\ e_1^n\ \mathrm{else}\ e_2^n$
$(\mathrm{let}\, x : \tau = e_1\, \mathrm{in}\, e_2)$	$(\lambda x : T\tau^n . e_2^n)\ e_1^n$
$\mu x : \tau . e$	$Y\ \tau^n\ (\lambda x : T\tau^n . e^n)$
$\lambda x : \tau . e$	$[\lambda x : T\tau^n . e^n]_T$
$e_1\ e_2$	$\mathrm{let}_T\, f \Leftarrow e_1^n\, \mathrm{in}\, f\ e_2^n$
(e_1, e_2)	$[(e_1^n, e_2^n)]_T$
$\pi_i\ e$	$\mathrm{let}_T\, x \Leftarrow e^n\, \mathrm{in}\, \pi_i\ x$

Fig. 2. CBN translation of Haskell

that a term e has type τ in the typing context Γ. In denotational semantics one is interested in interpreting only well-formed terms (since programs rejected by a type-checker are not allowed to run), thus we want to define a translation mapping well-formed terms $\Gamma \vdash_{PL} e : \tau$ of the programming language into well-formed terms $\Gamma \vdash_{ML} e : \tau$ of the metalanguage (with computational types). More precisely, we define a translation $_^n$ by induction on types τ and raw terms e, called the **CBN translation** (see Figure 2). When $\{x_i : \tau_i | i \in m\} \vdash_{PL} e : \tau$ is a well-formed term of Haskell, one can show that $\{x_i : T\tau_i^n | i \in m\} \vdash_{ML} e^n : T\tau^n$ is a well-formed term of the metalanguage with computational types. The signature Σ for defining the CBN translation of Haskell consists of

- $Y : \forall X : \bullet . (TX \to TX) \to TX$, a (least) fix-point combinator
- a signature for the datatype of integers.

Remark 21. The key feature of the CBN translation is that variables in the programming languages are translated into variables ranging over computational types. Another important feature is the translation of types, which basically guides (in combination with operational considerations) the translation of terms.

Exercise 22. Extend Haskell with polymorphism, as in 2nd-order λ-calculus, i.e.

$\tau \in Type_{Haskell} ::= X \mid \ldots \mid \forall X : \bullet . \tau \qquad e \in Exp_{Haskell} ::= \ldots \mid \Lambda X : \bullet . e \mid e\ \tau$

There is a problem to extend the CBN translation to polymorphic types. To overcome the problem assume that computational types commutes with polymorphism, i.e. the following map is an iso

$$c \ : \ T(\forall X : \bullet.\tau) \longmapsto \ \Lambda X : \bullet.\mathrm{let}_T \, x \Leftarrow c \, \mathrm{in} \, [x \ X]_T \ : \ \forall X : \bullet.T\tau$$

In realizability models several monads (e.g. lifting) satisfy this property, indeed the isomorphism is often an identity.

Algol Translation Some CBN languages (including Algol and PCF) allow computational effects only at base types. Computational types play a limited role in structuring the denotational semantics of these languages, nevertheless it is worth to compare the translation of such languages with that of Haskell. We consider an idealized-Algol with a fixed set of locations. Syntactically it is an extension of (simple) Haskell with three base types: Loc for integer locations, Int for integer expressions, and Cmd for commands. In Algol-like languages a location is often identified with a pair consisting of an expression and an *acceptor*, i.e. $\mathsf{Loc} \equiv (\mathsf{Int}, \mathsf{Int} \to \mathsf{Cmd})$.

$$\begin{aligned}
\tau \in Type_{Algol} ::= {} & \mathsf{Loc} \mid \mathsf{Int} \mid \mathsf{Cmd} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
e \in Exp_{Algol} ::= {} & x \mid \mathsf{l} \quad \text{location} \\
& \mid \mathsf{n} \mid e_0 + e_1 \mid !e \quad \text{contents of a location} \\
& \mathsf{if0} \ e_0 \ \mathsf{then} \ e_1 \ \mathsf{else} \ e_2 \\
& \mathsf{skip} \mid e_0 := e_1 \quad \text{null and assignment commands} \\
& e_0; e_1 \quad \text{sequential composition of commands} \\
& (\mathrm{let} \, x : \tau = e_1 \, \mathrm{in} \, e_2) \mid \mu x : \tau.e \mid \lambda x : \tau.e \mid e_1 \ e_2 \mid (e_1, e_2) \mid \pi_i \ e
\end{aligned}$$

The **Algol translation** $_^a$ (see Figure 3) is defined by induction on types τ and raw terms e. When $\{x_i : \tau_i | i \in m\} \vdash_{PL} e : \tau$ is a well-formed term of Algol, one can show that $\{x_i : \tau_i^a | i \in m\} \vdash_{ML} e^a : \tau^a$ is a well-formed term of the metalanguage with computational types.

Remark 23. The Algol translation seems to violate a key principle, namely that the translation of a program should have computational type. But in Algol valid programs are expected to be terms of base type, and the Algol translation indeed maps base types to computational types. More generally, one should observe that the Algol translation maps Algol types in (carriers of) T-algebras. Indeed T-algebras for a (strong) monads are closed under (arbitrary) products and exponentials, more precisely: $A_1 \times A_2$ is the carrier of a T-algebra whenever A_1 and A_2 are, and B^A is the carrier of a T-algebra whenever B is [EXERCISE: prove these facts in the metalanguage]. The T-algebra structure on the translation of types is used for defining the translation of terms, namely to extend the let and fix-point combinator from computational types to T-algebras:

$$- \ {}^*\mathrm{let} \ \frac{\Gamma \vdash e_1 : T\tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : U(\alpha : T\tau_2 \to \tau_2)}{\Gamma \vdash *\mathrm{let}_T \, x \Leftarrow e_1 \, \mathrm{in} \, e_2 \stackrel{\Delta}{=} \alpha(\mathrm{let}_T \, x \Leftarrow e_1 \, \mathrm{in} \, [e_2]_T) : U(\alpha : T\tau_2 \to \tau_2)}$$

$\tau \in Type_{Algol}$	$\tau^a \in Type$
Loc	TLoc
Int	TInt
Cmd	$T1$
$\tau_1 \to \tau_2$	$\tau_1^a \to \tau_2^a$
$\tau_1 \times \tau_2$	$\tau_1^a \times \tau_2^a$

$e \in Exp_{Algol}$	$e^a \in Exp$
x	x
n	$[\![\mathsf{n}]\!]_T$
$e_0 + e_1$	$\mathrm{let}_T\ x_0, x_1 \Leftarrow e_0^a, e_1^a\ \mathrm{in}\ [\![x_0 + x_1]\!]_T$
if0 e_0 then e_1 else e_2	$*\mathrm{let}_T\ x \Leftarrow e_0^a\ \mathrm{in}\ \mathrm{if}\ x = 0\ \mathrm{then}\ e_1^a\ \mathrm{else}\ e_2^a$
$(\mathrm{let}\ x : \tau = e_1\ \mathrm{in}\ e_2)$	$(\lambda x : \tau^a.e_2^a)\ e_1^a$
$\mu x : \tau.e$	$*Y\ \tau^a\ (\lambda x : \tau^a.e^a)$
$\lambda x : \tau.e$	$\lambda x : \tau^a.e^a$
$e_1\ e_2$	$e_1^a\ e_2^a$
(e_1, e_2)	(e_1^a, e_2^a)
$\pi_i\ e$	$\pi_i\ e^a$
|	$[\![\mathsf{I}]\!]_T$
$!e$	$\mathrm{let}_T\ l \Leftarrow e^a\ \mathrm{in}\ get\ l$
skip	$[\![()]\!]_T$
$e_0 := e_1$	$\mathrm{let}_T\ l, n \Leftarrow e_0^a, e_1^a\ \mathrm{in}\ set\ l\ n$
$e_0; e_1$	$\mathrm{let}_T\ _ \Leftarrow e_0^a\ \mathrm{in}\ e_1^a$

Fig. 3. Algol translation

$$- \ {}^*\mathrm{Y} \ \frac{\Gamma, x : \tau \vdash e : U(\alpha : T\tau \to \tau)}{\Gamma \vdash {}^*Y \ \tau \ (\lambda x : \tau.e) \stackrel{\Delta}{=} \alpha(Y \ \tau \ (\lambda c : T\tau.[e[x := \alpha \ c]]_T) : U(\alpha : T\tau \to \tau)}$$

The Algol translation suggests to put more emphasis on T-algebras. Indeed, [Lev99] has proposed a metalanguage for monads with two classes of types: value types interpreted by objects in $\mathcal{C}$, and computation types interpreted by objects in $\mathcal{C}^T$.

The signature Σ for defining the Algol translation consists of

- $Y : \forall X : \bullet.(TX \to TX) \to TX$, like for the Haskell translation
- a signature for the datatype of integers, like for the Haskell translation
- a type Loc of locations, with a fixed set of constants $\mathsf{l} : \mathsf{Loc}$, and operations $get : \mathsf{Loc} \to T\mathsf{Int}$ and $set : \mathsf{Loc} \to \mathsf{Int} \to T1$ to get/store an integer from/into a location.

Remark 24. In Algol expressions and commands have different computational effects, namely: expressions can only read the state, while commands can also modify the state. Therefore, one would have to consider two monads, $T_{sr}A = A^S_\perp$ for state reading computations and $T_{se}A = (A \times S)^S_\perp$ for computations with side-effects, and a monad morphism from T_{sr} to T_{se}.

CBV Translation: SML We consider a simple fragment of SML with integer locations. Syntactically the language is a minor variation of idealized Algol, more precisely: Cmd is replaced by Unit and skip by $()$, sequential composition of commands has been removed (because definable), recursive definitions are restricted to functional types.

$$\begin{aligned}
\tau \in Type_{SML} &::= \mathsf{Loc} \mid \mathsf{Int} \mid \mathsf{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
e \in Exp_{SML} &::= x \mid \mathsf{l} \mid \mathsf{n} \mid e_0 + e_1 \mid !e \mid \mathsf{if0} \ e_0 \ \mathsf{then} \ e_1 \ \mathsf{else} \ e_2 \mid () \mid e_0 := e_1 \mid \\
&\quad (\mathsf{let} \, x : \tau = e_1 \, \mathsf{in} \, e_2) \mid \mu f : \tau_1 \to \tau_2.\lambda x : \tau_1.e \mid \\
&\quad \lambda x : \tau.e \mid e_1 \ e_2 \mid (e_1, e_2) \mid \pi_i \ e
\end{aligned}$$

The **CBV translation** $_^v$ (see Figure 4) is defined by induction on types τ and raw terms e. When $\{x_i : \tau_i | i \in m\} \vdash_{PL} e : \tau$ is a well-formed term of SML, one can show that $\{x_i : \tau_i^v | i \in m\} \vdash_{ML} e^v : T\tau^v$ is a well-formed term of the metalanguage with computational types. The signature Σ for defining the CBV translation is the same used for defining the Algol translation.

Exercise 25. So far we have not said how to interpret the metalanguages used as target for the various translations. Propose interpretations of the metalanguages in the category of cpos: first choose a monad for interpreting computational types, then explain how the other symbols in the signature Σ should be interpreted.

Exercise 26. The translations considered so far allow to validate equational laws for the programming languages, by deriving the translation of the equational

$\tau \in Type_{SML}$	$\tau^v \in Type$
Loc	Loc
Int	Int
Unit	1
$\tau_1 \to \tau_2$	$\tau_1^v \to T\tau_2^v$
$\tau_1 \times \tau_2$	$\tau_1^v \times \tau_2^v$

$e \in Exp_{SML}$	$e^v \in Exp$
x	$[x]_T$
n	$[\mathsf{n}]_T$
$e_0 + e_1$	$\mathrm{let}_T\, x_0, x_1 \Leftarrow e_0^v, e_1^v \,\mathrm{in}\, [x_0 + x_1]_T$
if0 e_0 then e_1 else e_2	$\mathrm{let}_T\, x \Leftarrow e_0^v \,\mathrm{in}\, \mathrm{if}\ x = 0\ \mathrm{then}\ e_1^v\ \mathrm{else}\ e_2^v$
$(\mathrm{let}\, x : \tau = e_1 \,\mathrm{in}\, e_2)$	$\mathrm{let}_T\, x \Leftarrow e_1^v \,\mathrm{in}\, e_1^v$
$\mu f : \tau_1 \to \tau_2 . \lambda x : \tau_1 . e$	$*Y\ (\tau_1 \to \tau_2)^v\ (\lambda f : (\tau_1 \to \tau_2)^v . \lambda x : \tau_1^v . e^v)$
$\lambda x : \tau . e$	$[\lambda x : \tau^v . e^v]_T$
$e_1\ e_2$	$\mathrm{let}_T\, f, x \Leftarrow e_1^v, e_2^v \,\mathrm{in}\, f\ x$
(e_1, e_2)	$\mathrm{let}_T\, x_1, x_2 \Leftarrow e_1^v, e_2^v \,\mathrm{in}\, [(x_1, x_2)]_T$
$\pi_i\ e$	$\mathrm{let}_T\, x \Leftarrow e^v \,\mathrm{in}\, [\pi_i\ x]_T$
!	$[!]_T$
$!e$	$\mathrm{let}_T\, l \Leftarrow e^v \,\mathrm{in}\, get\ l$
()	$[()]_T$
$e_0 := e_1$	$\mathrm{let}_T\, l, n \Leftarrow e_0^v, e_1^v \,\mathrm{in}\, set\ l\ n$

Fig. 4. CBV translation of SML

laws in the metalanguage. Say whether β and η for functional types, i.e. $(\lambda x : \tau_1 . e_2)\ e_1 = e_2[x := e_1] : \tau_2$ and $(\lambda x : \tau_1 . e\ x) = e : \tau_1 \to \tau_2$ with $x \notin \mathrm{FV}(e)$, are valid in Haskell, Algol or SML. If they are not valid suggest weaker equational laws that can be validate. This exercise indicates that one should be careful to transfer reasoning principle for the λ-calculus to functional languages.

Exercise 27. Consider Haskell with integer locations, and extend the CBN translation accordingly. Which signature Σ should be used?

Exercise 28. In SML one can create new locations using the construct ref e. Consider this extension of SML, and extend the CBV translation accordingly. Which signature Σ and monad T in the category of cpos should be used?

Exercise 29. Consider SML with locations of any type, and extend the CBV translation accordingly. Which signature Σ should be used (you may find convenient to assume that the metalanguage includes higher-order λ-calculus)? It is very difficult to find monads able to interpret such a metalanguage.

3.2 Incremental Approach and Monad Transformers

The monadic approach to denotational semantics has a caveat. When the programming language PL is complex, the signature Σ identified by the monadic

approach can get fairly large, and the translation of $ML_T(\Sigma)$ into ML may become quite complicated.

One can alleviate the problem by adopting an **incremental approach** in defining the translation of $ML_T(\Sigma)$ into ML. The basic idea is to adapt to this setting the techniques and modularization facilities advocated for formal software development, in particular the desired translation of $ML_T(\Sigma)$ into ML corresponds to the implementation of an abstract datatype (in some given language). In an incremental approach, the desired implementation is obtained by a sequence of steps, where each step constructs an implementation for a more complex datatype from an implementation for a simpler datatype.

> Haskell constructor classes (and to a less extend SML modules) provide a very convenient setting for the incremental approach (see [LHJ95]): the type inference mechanism allows concise and readable definitions, while type-checking detects most errors. What is missing is only the ability to express and validate (equational) properties, which would require extra features typical of Logical Frameworks (see [Mog97]).

To make the approach viable, we need a collection of self-contained parameterized *polymorphic* modules with the following features:

- they should be polymorphic, i.e. for any signature Σ (or at least for a wide range of signatures) the module should take an implementation of Σ and construct an implementation of $\Sigma + \Sigma_{new}$, where Σ_{new} is fixed
- they could be parametric, i.e. the construction and the signature Σ_{new} may depend on parameters of some fixed signature Σ_{par}.

The polymorphic requirement can be easily satisfied, when one can implement Σ_{new} without changing the implementation of Σ (this is often the case in software development). However, the constructions we are interested in are not *persistent*, since they involve a re-implementation of computational types, and consequently of Σ. The translations we need to consider are of the form

$$I : ML_T(\Sigma_{par} + \Sigma + \Sigma_{new}) \to ML_T(\Sigma_{par} + \Sigma)$$

where Σ_{new} are the new symbols defined by I, Σ the old symbols *redefined* by I, and Σ_{par} the parameters of the construction (which are unaffected by I). In general I can be decomposed in

- a translation $I_{new} : ML_T(\Sigma_{par} + \Sigma_{new}) \to ML_T(\Sigma_{par})$ defining the new symbols (in Σ_{new}) and redefining computational types,
- translations $I_{op} : ML_T(\Sigma_{op}) \to ML_T(\Sigma_{par} + \Sigma_{op})$ redefining an old symbol *op* in *isolation* (consistently with the redefinition of computational types), for each possible type of symbol one may have in Σ.

Recently [Fil99] has proposed a more flexible approach, which uses metalanguages with several monads T_i (rather than only one), and at each step it

introduces a new monad T' and new operations (defined in term of the pre-existing ones), without changing the meaning of the old symbols. Therefore, one is considering *definitional extensions*, i.e. translations of the form

$$I : ML_{T',T_{i\in n}}(\Sigma_{old} + \Sigma'_{new}) \to ML_{T_{i\in n}}(\Sigma_{old})$$

which are the identity on $ML_{T_{i\in n}}(\Sigma_{old})$. In Filinski's approach one can use the translations I_{new} and I_{op}, whenever possible, and more ad hoc definitions otherwise. In fact, when Filinski introduces a new monad T', he introduces also two operations called monadic reflection and reification

$$\mathsf{reflect} : \forall X : \bullet.\tau \to T'X \qquad \mathsf{reify} : \forall X : \bullet.T'X \to \tau$$

that establish a bijection between $T'X$ and its implementation τ (i.e. a type in the pre-existing language). Therefore, one can define operations related to T' by moving back and forth between T' and its implementation (as done in the case of operations defined on an abstract datatype).

Semantically a **monad transformer** is a function $F : |Mon(\mathcal{C})| \to |Mon(\mathcal{C})|$ mapping monads (over a category $\mathcal{C}$) to monads. We are interested in monad transformers for *adding computational effects*, therefore we require that for any monad T there should be a monad morphism $in_T : T \to FT$. It is often the case that F is a functor on $Mon(\mathcal{C})$, and in becomes a natural transformation from $\mathrm{id}_{Mon(\mathcal{C})}$ to F. Syntactically a **monad transformer** is a translation

$$I_F : ML_{T',T}(\Sigma_{par}) \to ML_T(\Sigma_{par})$$

which is the identity on $ML_T(\Sigma_{par})$. In other words we express the new monad T' in terms of the old monad T (and the parameters specified in Σ_{par}). In the sequel we describe (in a higher-order λ-calculus) several monad transformers corresponding to the addition of a new computational effect, more precisely we define

- the new monad T', and the monad morphism $in : T \to T'$
- the operations on T'-computations associated to the new computational effect
- an operation $op' : \forall X : \bullet.A \to (B \to T'X) \to T'X$ *extending* to T'-computations a pre-existing operation $op : \forall X : \bullet.A \to (B \to TX) \to TX$ on T-computations.

Monad Transformer I_{se} for Adding Side-Effects

- signature Σ_{par} for parameters
  states $S : \bullet$
- signature Σ_{new} for new operations
  lookup $lkp' : T'S$
  update $upd' : S \to T'1$

– definition of new monad T' and monad morphism $in : T \to T'$

$$T'X \triangleq S \to T(X \times S)$$
$$[x]_{T'} \triangleq \lambda s.[(x,s)]_T$$
$$\text{let}_{T'}\ x \Leftarrow c \,\text{in}\, f\ x \triangleq \lambda s.\text{let}_T\ (x,s') \Leftarrow c\ s \,\text{in}\, f\ x\ s'$$
$$in\ X\ c \triangleq \lambda s.\text{let}_T\ x \Leftarrow c \,\text{in}\, [(x,s)]_T$$

definition of new operations

$$lkp' \triangleq \lambda s.[(s,s)]_T$$
$$upd'\ s \triangleq \lambda s'.[(*,s)]_T$$

extension of old operation

$$op'\ X\ a\ f \triangleq \lambda s.op\ (X \times S)\ a\ (\lambda b.f\ b\ s)$$

Remark 30. The operations lkp' and upd' do not fit the format for op. However, given an operation $*op : A \to TB$ one can define an operation $op : \forall X : \bullet.A \to (B \to TX) \to TX$ in the right format by taking $op\ X\ a\ f \triangleq \text{let}_T\ b \Leftarrow *op\ a \,\text{in}\, f\ b$.

Monad Transformer I_{ex} for Adding Exceptions

– signature Σ_{par} for parameters
exceptions $E : \bullet$
– signature Σ_{new} for new operations
raise $raise' : \forall X : \bullet.E \to T'X$
handle $handle' : \forall X : \bullet.(E \to T'X) \to T'X \to T'X$
– definition of new monad T' and monad morphism $in : T \to T'$

$$T'X \triangleq T(X + E)$$
$$[x]_{T'} \triangleq [\text{inl}\ x]_T$$
$$\text{let}_{T'}\ x \Leftarrow c \,\text{in}\, f\ x \triangleq \text{let}_T\ u \Leftarrow c \,\text{in}\, (\text{case}\, u \,\text{of}\, x \Rightarrow f\ x \mid n \Rightarrow [\text{inr}\ n]_T)$$
$$in\ X\ c \triangleq \text{let}_T\ x \Leftarrow c \,\text{in}\, [\text{inl}\ x]_T$$

definition of new operations

$$raise'\ X\ n \triangleq [\text{inr}\ n]_T$$
$$handle'\ X\ f\ c \triangleq \text{let}_T\ u \Leftarrow c \,\text{in}\, (\text{case}\, u \,\text{of}\, x \Rightarrow [\text{inl}\ x]_T \mid n \Rightarrow f\ n)$$

extension of old operation

$$op'\ X\ a\ f \triangleq op\ (X + E)\ a\ f$$

Remark 31. In this case the definition of op' is particularly simple, and one can show that the same definition works for extending a more general type of operations.

Monad Transformer I_{co} for Adding Complexity

– signature Σ_{par} for parameters
monoid $M : \bullet$
$1 : M$
$* : M \to M \to M$ (we use infix notation for $*$)

to prove that T' is a monad, we should add axioms saying that $(M, 1, *)$ is a monoid

- signature Σ_{new} for new operations
  cost $tick' : M \to T'1$
- definition of new monad T' and monad morphism $in : T \to T'$

$$T'X \triangleq T(X \times M)$$
$$[x]_{T'} \triangleq [(x, 1)]_T$$
$$\text{let}_{T'}\ x \Leftarrow c \,\text{in}\, f\ x \triangleq \text{let}_T\ (x, m) \Leftarrow c \,\text{in}\, (\text{let}_T\ (y, n) \Leftarrow f\ x \,\text{in}\, [(y, m * n)]_T)$$
$$in\ X\ c \triangleq \text{let}_T\ x \Leftarrow c \,\text{in}\, [(x, 1)]_T$$

definition of new operations

$$tick'\ m \triangleq [(*, m)]_T$$

extension of old operation

$$op'\ X\ a\ f \triangleq op\ (X \times M)\ a\ f$$

Monad Transformer I_{con} for Adding Continuations

- signature Σ_{par} for parameters
  results $R : \bullet$
- signature Σ_{new} for new operations
  abort $abort' : \forall X : \bullet. R \to T'X$
  call-cc $callcc' : \forall X, Y : \bullet. ((X \to T'Y) \to T'X) \to T'X$
- definition of new monad T' and monad morphism $in : T \to T'$

$$T'X \triangleq (X \to TR) \to TR$$
$$[x]_{T'} \triangleq \lambda k. k\ x$$
$$\text{let}_{T'}\ x \Leftarrow c \,\text{in}\, f\ x \triangleq \lambda k. c\ (\lambda x. f\ x\ k)$$
$$in\ X\ c \triangleq \lambda k. \text{let}_T\ x \Leftarrow c \,\text{in}\, k\ x$$

definition of new operations

$$abort'\ X\ r \triangleq \lambda k. [r]_T$$
$$callcc'\ X\ Y\ f \triangleq \lambda k. f\ (\lambda x. \lambda k'. [k\ x]_T)\ k$$

extension of old operation

$$op'\ X\ a\ f \triangleq \lambda k. op\ R\ a\ (\lambda b. f\ b\ k)$$

Remark 32. The operation $callcc'$ does not fit the specified format for an old operation, and there is no way to massage it into such format. Unlike the others monad transformers, I_{con} does not extend to a functor on $Mon(\mathcal{C})$.

Exercise 33. For each of the monad transformer, prove that T' is a monad. Assume that T is a monad, and use the equational axioms for higher-order λ-calculus with sums and products, including η-axioms.

Exercise 34. For each of the monad transformer, define a fix-point combinator for the new computational types $Y' : \forall X : \bullet. (T'X \to T'X) \to T'X$ given a fix-point combinator for the old computational types $Y : \forall X : \bullet. (TX \to TX) \to TX$. In some cases one should use the derived fix-point combinator $*Y$ for carriers of T-algebras.

Exercise 35. Define a monad transformer I_{sr} for state-readers, i.e. $T'X \stackrel{\Delta}{=} S \to TX$. What could be Σ_{new}? Define a monad morphism from T_{sr} to T_{se}.

Exercise 36. Check which monad transformers commutes (up to isomorphism). For instance, I_{se} and I_{ex} do not commute, more precisely one gets

- $T_{se+ex}X = S \to T((X + E)\times S)$ when adding first side-effects and then exceptions
- $T_{ex+se}X = S \to T((X\times S) + E)$ when adding first exceptions and then side-effects

Exercise 37. For each of the monad transformer, identify equational laws for the new operations specified in Σ_{new}, and show that such laws are validated by the translation. For instance, I_{se} validates the following equations:

$$
\begin{array}{l}
s : S \vdash \mathrm{let}_{T'}\ * \Leftarrow upd'\ s\ \mathrm{in}\ lkp' = \mathrm{let}_{T'}\ [s]_{T'} \Leftarrow upd'\ s\ \mathrm{in}\ : T'S \\
s, s' : S \vdash \mathrm{let}_{T'}\ * \Leftarrow upd'\ s\ \mathrm{in}\ upd'\ s' = upd'\ s' : T'1 \\
s : S \vdash \mathrm{let}_{T'}\ s \Leftarrow lkp'\ \mathrm{in}\ upd'\ s = [*]_{T'} : T'1 \\
X : \bullet, c : T'X \vdash \mathrm{let}_{T'}\ s \Leftarrow lkp'\ \mathrm{in}\ c = c : T'X
\end{array}
$$

Exercise 38 (Semantics of Effects). Given a monad T over the category of sets:

- Define predicates for $c \in T'X \stackrel{\Delta}{=} S \to T(X\times S)$ corresponding to the properties "c does not read from S" and "c does not write in S".
  notice that such predicates are *extensional*, therefore a computation that reads the state and then rewrites it unchanged, is equivalent to a computation that ignores the state.
- Define a predicate for $c \in T'X \stackrel{\Delta}{=} T(X + E)$ corresponding to the property "c does not raise exceptions in E".

4 Monads in Haskell

So far we have focussed on applications of monads in denotational semantics, but since Wadler's influential papers in the early 90s they have also become part of the toolkit that Haskell programmers use on a day to day basis. Indeed, monads have proven to be so useful in practice that the language now includes extensions specifically to make programming with them easy. In the next few sections we will see how monads are represented in Haskell, look at some of their applications, and try to explain why they have had such an impact.

4.1 Implementing Monads in Haskell

The representation of monads in Haskell is based on the *Kleisli triple* formulation: recall Definition 1.4:

> A **Kleisli triple** over a category $\mathcal{C}$ is a triple $(T, \eta, _^*)$, where $T : |\mathcal{C}| \to |\mathcal{C}|$, $\eta_A : A \to TA$ for $A \in |\mathcal{C}|$, $f^* : TA \to TB$ for $f : A \to TB$ and the following equations hold: ...

In Haskell, T corresponds to a parameterised type, η is called `return`, and $_^*$ is called `>>=`. This would suggest the following types:

```
return :: a -> T a
(>>=) :: (a -> T b) -> (T a -> T b)
```

where `a` and `b` are Haskell type variables, so that these types are polymorphic. But notice that we can consider `>>=` to be a curried function of *two* arguments, with types `(a -> T b)` and `T a`. In practice it is convenient to reverse these arguments, and instead give `>>=` the type

```
(>>=) :: T a -> (a -> T b) -> T b
```

Now the metalanguage notation let $x \Leftarrow e_1$ in e_2 can be conveniently expressed as

```
e1 >>= \x -> e2
```

(where `\x -> e` is Haskell's notation for $\lambda x.e$). Intuitively this binds `x` to the result of `e1` in `e2`; with this in mind we usually pronounce "`>>=`" as "bind".

Example 39. The monad of **partiality** can be represented using the built-in Haskell type

```
data Maybe a = Just a | Nothing
```

This defines a parameterised type `Maybe`, whose elements are `Just x` for any element `x` of type `a` (representing a successful computation), or `Nothing` (representing failure).

The monad operators can be implemented as

```
return a = Just a

m >>= f = case m of
            Just a -> f a
            Nothing -> Nothing
```

and failure can be represented by

```
failure = Nothing
```

As an example of an application, a division function which operates on possibly failing integers can now be defined as

```
divide :: Maybe Int -> Maybe Int -> Maybe Int
divide a b = a >>= \m ->
             b >>= \n ->
             if n==0 then failure
                     else return (a `div` b)
```

Try unfolding the calls of `>>=` in this definition to understand the gain in clarity that using monadic operators brings.

Example 40. As a second example, we show how to implement the monad of **side-effects** in Haskell. This time we will need to define a new type, `State s a`, to represent computations producing an `a`, with a side-effect on a state of type `s`. Haskell provides three ways to define types:

```
type State s a = s -> (s,a)
newtype State s a = State (s -> (s,a))
data State s a = State (s -> (s,a))
```

The first alternative declares a *type synonym*: `State s a` would be in every respect equivalent to the type `s -> (s,a)`. This would cause problems later: since many monads are represented by functions, it would be difficult to tell *just from the type* which monad we were talking about.

The second alternative declares `State s a` to be a new type, different from all others, but isomorphic to `s -> (s,a)`. The elements of the new type are written `State f` to distinguish them from functions. (There is no need for the tag used on elements to have the same name as the type, but it is often convenient to use the same name for both).

The third alternative also declares `State s a` to be a new type, with elements of the form `State f`, but in contrast to `newtype` the `State` constructor is lazy: that is, `State` $\perp$ and $\perp$ are different values. This is because `data` declarations create lifted sum-of-product types, and even when the sum is trivial it is still lifted. Thus `State s a` is *not* isomorphic to `s -> (s,a)` — it has an extra element — and values of this type are more costly to manipulate as a result.

We therefore choose the second alternative. The monad operations are now easy to define:

```
return a = State (\s -> (s,a))

State m >>= f = State (\s -> let (s',a) = m s
                                 State m' = f a
                             in m' s')
```

The state can be manipulated using

```
readState :: State s s
readState = State (\s -> (s,s))

writeState :: s -> State s ()
writeState s = State (\_ -> (s,()))
```

For example, a function to increment the state could be expressed using these functions as

```
increment :: State Int ()
increment = readState >>= \s ->
            writeState (s+1)
```

4.2 The Monad Class: Overloading return and Bind

Haskell programmers make use of many different monads; it would be awkward if `return` and `>>=` had to be given different names for each one. To avoid this, we use *overloading* so that the same names can be used for every monad.

Overloading in Haskell is supported via the *class system*: overloaded names are introduced by defining a *class* containing them. A class is essentially a signature, with a different implementation for each type. The monad operations are a part of a class `Monad`, whose definition is found in Haskell's standard prelude:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Here the class parameter `m` ranges over *parameterised* types; read the declaration as "A parameterised type `m` is a `Monad` if it supports implementations of `return` and `>>=` with the given types".

Implementations of these operations are provided by making a corresponding *instance* declaration, for example:

```
instance Monad Maybe where
  return a = Just a
  m >>= f = case m of
              Just a -> f a
              Nothing -> Nothing
```

which corresponds to the definition of the `Maybe` monad given earlier. For the monad of side-effects, we write

```
instance Monad (State s) where
  return a = State (\s -> (s,a))
  State m >>= f = State (\s -> let (s',a) = m s
                                   State m' = f a
                               in m' s')
```

Notice that although we defined the type `State` with two parameters, and the `Monad` class requires a type with one parameter, Haskell allows us to create the type we need by *partially applying* the `State` type to one parameter: types with many parameters are 'curried'. Indeed, we chose the order of the parameters in the definition of `State` with this in mind.

Now when the monadic operators are applied, the type at which they are used determines which implementation is invoked. This is why we were careful to make `State` a *new* type above.

A great advantage of overloading the monad operators is that it enables us to write code which works with *any* monad. For example, we could define a function which combines two monadic computations producing integers into a computation of their sum:

```
addM a b = a >>= \m ->
           b >>= \n ->
           return (m+n)
```

Since nothing in this definition is specific to a particular monad, we can use this function with any: `addM (Just 2) (Just 3)` is `Just 5`, but we could also use `addM` with the `State` monad. The type assigned to `addM` reflects this, it is[1]

```
addM :: (Monad m) => m Int -> m Int -> m Int
```

The "`(Monad m) =>`" is called a *context*, and restricts the types which may be substituted for `m` to instances of the class `Monad`.

Although `addM` is perhaps too specialised to be really useful, we can derive a very useful higher-order function by generalising over `+`. Indeed, Haskell's standard `Monad` library provides a number of such functions, such as

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
sequence :: Monad m => [m a] -> m [a]
```

With these definitions,

```
addM = liftM2 (+)
```

Programming with monads is greatly eased by such a library.

Exercise 41. Give a definition of `sequence`. The intention is that each computation in the list is executed in turn, and a list made of the results.

[1] Actually type inference produces an even more general type, since the arithmetic is also overloaded, but we will gloss over this.

Finally, Haskell provides syntactic sugar for `>>=` to make monadic programs more readable: the `do`-notation. For example, the definition of `addM` above could equivalently be written as

```
addM a b = do m <- a
              n <- b
              return (m+n)
```

The `do`-notation is defined by

```
do e          = e

do x <- e     = e >>= (\x -> do c)
   c

do e          = e >>= (\_ -> do c)
   c
```

Applying these rules to the definition of `addM` above rewrites it into the form first presented. The `do`-notation is simply a shorthand for bind, but does make programs more recognisable, especially for beginners.

Example 42. As an example of monadic programming, consider the problem of decorating the leaves of a tree with unique numbers. We shall use a parameterised tree type

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

and define a function

```
unique :: Tree a -> Tree (a,Int)
```

which numbers the leaves from 1 upwards in left-to-right order. For example,

```
unique (Bin (Bin (Leaf 'a') (Leaf 'b')) (Leaf 'c'))
 = Bin (Bin (Leaf ('a',1)) (Leaf ('b',2))) (Leaf ('c',3))
```

Intuitively we think of an integer state which is incremented every time a leaf is encountered: we shall therefore make use of the `State` monad to define a function

```
unique' :: Tree a -> State Int (Tree (a,Int))
```

First we define a function to increment the state,

```
tick :: State Int Int
tick = do n <- readState
          writeState (n+1)
          return n
```

and then the definition of `unique'` is straightforward:

```
unique' (Leaf a) = do n <- tick
                      return (Leaf (a,n))
unique' (Bin t1 t2) = liftM2 Bin (unique' t1) (unique' t2)
```

Notice that we use `liftM2` to apply the two-argument function `Bin` to the results of labelling the two subtrees; as a result the notational overhead of using a monad is very small.

Finally we define `unique` to invoke the monadic function and supply an initial state:

```
unique t = runState 1 (unique' t)

runState s (State f) = snd (f s)
```

It is instructive to rewrite the `unique` function directly, without using a monad — explicit state passing in the recursive definition clutters it significantly, and creates opportunities for errors that the monadic code completely avoids.

5 Applying Monads

So far we have shown how monads are represented in Haskell, and how the language supports their use. But what are monads used *for*? Why have they become so prevalent in Haskell programs? In this section we try to answer these questions.

5.1 Input/Output: the Killer Application

Historically, input/output has been awkward to handle in *purely* functional languages. The same applies to foreign function calls: there is no way to *guarantee* that a function written in C, for example, does not have side effects, so calling it directly from a Haskell program would risk compromising Haskell's purely functional semantics.

Yet it is clear enough that input/output can be *modelled* in a purely functional way: we must just consider a program to be a function from the state of the universe before it is run, to the state of the universe afterwards. One possibility is to write the program in this way: every function depending on the external state would take the universe as a parameter, and every function modifying it would return a new universe as a part of its result. For example a program to copy one file to another might be written as

```
copy :: String -> String -> Universe -> Universe
copy from to universe =
  let contents = readFile from universe
      universe' = writeFile to contents universe
  in universe'
```

Such a program has a purely functional semantics, but is not easy to implement. Of course, we cannot really maintain several copies of the universe at the same time, and so 'functions' such as `writeFile` must be implemented by actually writing the new contents to the filestore. If the programmer then

accidentally or deliberately returns `universe` instead of `universe'` as the final result of his program, then the purely functional semantics is not correctly implemented. This approach has been followed in Clean though, using a linear type system to guarantee that the programmer manipulates universes correctly [BS96].

However, having seen monads we would probably wish to simplify the program above by using a `State` monad to manage the universe. By defining

```
type IO a = State Universe a
```

and altering the types of the primitives slightly to

```
readFile :: String -> IO String
writeFile :: String -> String -> IO ()
```

then we can rewrite the file copying program as

```
copy :: String -> String -> IO ()
copy from to = do contents <- readFile from
                  writeFile to contents
```

which looks almost like an imperative program for the same task[2].

This program is both purely functional and efficiently implementable: it is quite safe to write the output file destructively. However, there is still a risk that the programmer will define inappropriate operations on the `IO` type, such as

```
snapshot :: IO Universe
snapshot = State (\univ -> (univ, univ))
```

The solution is just to *make the* `IO` *type abstract* [JW93]! This does not change the semantics of programs, which remains purely functional, but it *does* guarantee that as long as all the primitive operations on the `IO` type treat the universe in a proper single-threaded way (which all operations implemented in imperative languages do), then so does any Haskell program which uses them.

Since the `IO` monad was introduced into Haskell, it has been possible to write Haskell programs which do input/output, call foreign functions directly, and yet still have a purely functional semantics. Moreover, these programs look very like ordinary programs in any imperative language. The contortions previously needed to achieve similar effects are not worthy of description here.

The reader may be wondering what all the excitement is about here: after all, it has been possible to write ordinary imperative programs in other languages for a very long time, including functional languages such as ML or Scheme; what is so special about writing them in Haskell? Two things:

[2] The main difference is that we read and write the entire contents of a file in one operation, rather than byte-by-byte as an imperative program probably would. This may seem wasteful of space, but thanks to lazy evaluation the characters of the input file are only actually read into memory when they are needed for writing to the output. That is, the space requirements are small and constant, just as for a byte-by-byte imperative program.

- Input/output can be combined cleanly with the other features of Haskell, in particular higher-order functions, polymorphism, and lazy evaluation. Although ML, for example, combines input/output with the first two, the ability to mix lazy evaluation cleanly with I/O is unique to Haskell with monads — and as the `copy` example shows, can lead to simpler programs than would otherwise be possible.
- Input/output is combined with a purely functional semantics. In ML, for example, *any* expression may potentially have side-effects, and transformations which re-order computations are invalid without an effect analysis to establish that the computations are side-effect free. In Haskell, no expression has side-effects, but some denote commands with effects; moreover, the potential to cause side-effects is evident in an expression's type. Evaluation order can be changed freely, but monadic computations cannot be reordered because the monad laws do not permit it.

Peyton-Jones' excellent tutorial [Pey01] covers this kind of monadic programming in much more detail, and also discusses a useful refinement to the semantics presented here.

5.2 Imperative Algorithms

Many algorithms can be expressed in a purely functional style with the same complexity as their imperative forms. But some efficient algorithms depend critically on destructive updates. Examples include the UNION-FIND algorithm, many graph algorithms, and the implementation of arrays with constant time access and modification. Without monads, Haskell cannot express these algorithms with the same complexity as an imperative language.

With monads, however, it is easy to do so. Just as the *abstract* `IO` monad enables us to write programs with a purely functional semantics, and give them an imperative implementation, so an abstract *state transformer* monad `ST` allows us to write purely functional programs which update the state destructively [LJ94][3]. Semantically the type `ST a` is isomorphic to `State -> (State,a)`, where `State` is a function from typed references (locations) to their contents. In the implementation, only one `State` ever exists, which is updated destructively in place.

Operations are provided to create, read, and write typed references:

```
newSTRef :: a -> ST (STRef a)
readSTRef :: STRef a -> ST a
writeSTRef :: STRef a -> a -> ST ()
```

Here `STRef a` is the type of a reference containing a value of type `a`. Other operations are provided to create and manipulate arrays.

The reason for introducing a *different* monad `ST`, rather than just providing these operations over the `IO` monad, is that destructive updates to variables in

[3] While the IO monad is a part of Haskell 98, the current standard [JHe+99], the ST monad is not. However, every implementation provides it in some form; the description here is based on the Hugs modules ST and LazyST [JRtYHG+99].

a program are not *externally visible* side-effects. We can therefore encapsulate these imperative effects using a new primitive

```
runST :: ST a -> a
```

which semantically creates a new `State`, runs its argument in it, and discards the final `State` before returning an `a` as its result. (A corresponding function `runIO` would not be implementable, because we have no way to 'discard the final `Universe`'). In the implementation of `runST`, `State`s are represented just by a collection of references stored on the heap; there is no cost involved in creating a 'new' one therefore. Using `runST` we can write pure (non-monadic) functions whose implementation uses imperative features internally.

Example 43. The depth-first search algorithm for graphs uses destructively updated marks to identify previously visited nodes and avoid traversing them again. For simplicity, let us represent graph nodes by integers, and graphs using the type

```
type Graph = Array Int [Int]
```

A graph is an array indexed by integers (nodes), whose elements are the list of successors of the corresponding node. We can record which nodes have been visited using an updateable array of boolean marks, and program the depth-first search algorithm as follows:

```
dfs g ns = runST (do marks <- newSTArray (bounds g) False
                     dfs' g ns marks)

dfs' g [] marks = return []
dfs' g (n:ns) marks =
  do visited <- readSTArray marks n
     if visited then dfs' g ns marks
       else do writeSTArray marks n True
               ns' <- dfs' g ((g!n)++ns) marks
               return (n:ns')
```

The function `dfs` returns a list of all nodes reachable from the given list of roots in depth-first order, for example:

```
dfs (array (1,4) [(1,[2,3]), (2,[4]), (3,[4]), (4,[1])]) =
  [1,2,4,3]
```

The *type* of the depth-first search function is

```
dfs :: Graph -> [Int] -> [Int]
```

It is a pure, non-monadic function which can be freely mixed with other non-monadic code.

Imperative features combine interestingly with lazy evaluation. In this example, the output list is produced lazily: the traversal runs only far enough to

produce the elements which are demanded. This is possible because, in the code above, `return (n:ns')` can produce a result *before* `ns'` is known. The recursive call of `dfs'` need not be performed until the value of `ns'` is actually needed[4]. Thus we can efficiently use `dfs` even for incomplete traversals: to search for the first node satisfying `p`, for example, we can use

```
head (filter p (dfs g roots))
```

safe in the knowledge that the traversal will stop when the first node is found.

King and Launchbury have shown how the lazy depth-first search function can be used to express a wide variety of graph algorithms both elegantly and efficiently [KL95].

The `ST` monad raises some interesting typing issues. Note first of all that its operations cannot be implemented in Haskell with the types given, even inefficiently! The problem is that we cannot represent an indexed collection of values with arbitrary types – if we tried to represent `State`s as functions from references to contents, for example, then all the contents would have to have the same type. A purely functional implementation would need *dependent types*, to allow the type of a reference's contents to depend on the reference itself. Thus the `ST` monad gives the Haskell programmer indirect access to dependent types, and indeed, sometimes other applications which require dependent types can be programmed in terms of `ST`.

Secondly, we must somehow prevent references created in one `State` being used in another — it would be hard to assign a sensible meaning to the result. This is done by giving the `ST` type an additional parameter, which we may think of as a 'state identifier': `ST s a` is the type of computations *on state* `s` producing an `a`. Reference types are also parameterised on the state identifier, so the types of the operations on them become:

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

These types guarantee that `ST` computations only manipulate references lying in 'their' `State`.

But what should the type of `runST` be? It is supposed to create a *new* `State` to run its argument in, but if we give it the type

```
runST :: ST s a -> a
```

then it will be applicable to *any* `ST` computation, including those which manipulate references in other `State`s. To prevent this, `runST` is given a *rank-2 polymorphic* type:

```
runST :: (forall s. ST s a) -> a
```

[4] Hugs actually provides two variations on the ST monad, with and without lazy behaviour. The programmer chooses between them by importing either ST or LazyST.

(and Hugs has been extended with rank-2 polymorphism just to make this possible). This type ensures that the argument of `runST` can safely be run in *any* `State`, in particular the new one which `runST` creates.

Example 44. The expression

```
runST (newSTRef 0)
```

is not well-typed. Since `newSTRef 0` has the type `ST s (STRef s Int)`, then `runST` would have to produce a result of type `STRef s Int` — but the scope of `s` does not extend over the type of the result.

Example 45. The expression

```
runST (do r<-newSTRef 0
          return (runST (readSTRef r)))
```

is not well-typed either, because the argument of the inner `runST` is not polymorphic — it depends on the state identifier of the outer one.

The inclusion of the `ST` monad and assignments in Haskell raises an interesting question: just what is a *purely* functional language? Perhaps the answer is: one in which assignment has a funny type!

5.3 Domain Specific Embedded Languages

Since the early days of functional programming, *combinator libraries* have been used to define succinct notations for programs in particular domains [Bur75]. There are combinator libraries for many different applications, but in this section we shall focus on one very well-studied area: parsing. A library for writing parsers typically defines a type `Parser a`, of parsers for values of type `a`, and combinators for constructing and invoking parsers. These might include

```
satisfy :: (Char -> Bool) -> Parser Char
```

to construct a parser which accepts a single character satisfying the given predicate,

```
(|||) :: Parser a -> Parser a -> Parser a
```

to construct a parser which accepts an input if *either* of its operands can parse it, and

```
runParser :: Parser a -> String -> a
```

to invoke a parser on a given input.

A parsing library must also include combinators to run parsers in sequence, and to build parsers which invoke functions to compute their results. Wadler realised that these could be provided by declaring the `Parser` type to be a monad [Wad92]. Further combinators can then be defined in terms of these basic ones, such as a combinator accepting a particular character,

```
literal :: Char -> Parser Char
literal c = satisfy (==c)
```

and a combinator for repetition,

```
many :: Parser a -> Parser [a]
many p = liftM2 (:) p (many p) ||| return []
```

which parses a list of any number of ps.

Given such a library, parsing programs can be written very succinctly. As an example, we present a function to evaluate arithmetic expressions involving addition and multiplication:

```
eval :: String -> Int
eval = runParser expr

expr = do t <- term
          literal '+'
          e <- expr
          return (t+e)
   ||| term

term = do c <- closed
          literal '*'
          t <- term
          return (c*t)
   ||| closed

closed = do literal '('
            e <- expr
            literal ')'
            return e
     ||| numeral

numeral = do ds <- many (satisfy isDigit)
             return (read ds)
```

With a good choice of combinators, the code of a parser closely resembles the grammar it parses[5][6]!

In recent years, a different view of such combinator libraries has become popular: we think of them as defining a *domain specific language* (DSL), whose

[5] In practice the resemblance would be a little less close: real parsers for arithmetic expressions are left-recursive, use a lexical analyser, and are written to avoid expensive backtracking. On the other hand, real parsing libraries provide more combinators to handle these features and make parsers even more succinct! See [HM98] for a good description.

[6] Notice how important Haskell's lazy evaluation is here: without it, these recursive definitions would not make sense!

constructions are the combinators of the library [Hud98]. With this view, this little parsing library defines a programming language with special constructions to accept a symbol and to express alternatives.

Every time a functional programmer designs a combinator library, then, we might as well say that he or she designs a domain specific programming language, integrated with Haskell. This is a useful perspective, since it encourages programmers to produce a modular design, with a clean separation between the semantics of the DSL and the program that uses it, rather than mixing combinators and 'raw' semantics willy-nilly. And since monads appear so often in programming language semantics, it is hardly surprising that they appear often in combinator libraries also!

We will return to the implementation of the parsing library in the next section, after a discussion of monad transformers.

6 Monad Transformers in Haskell

The Haskell programmer who makes heavy use of combinators will need to implement a large number of monads. Although it is perfectly possible to define a new type for each one, and implement `return` and `>>=` from scratch, it saves labour to construct monads systematically where possible. The *monad transformers* of section 3.2 offer an attractive way of doing so, as Liang, Hudak and Jones point out [LHJ95].

Recall the definition:

> A **monad transformer** is a function $F : |Mon(\mathcal{C})| \to |Mon(\mathcal{C})|$, i.e. a function mapping monads (over a category $\mathcal{C}$) to monads. We are interested in monad transformers for *adding computational effects*, therefore we require that for any monad T there should be a monad morphism $in_T : T \to FT$.

We represent monad transformers in Haskell by types parameterised on a monad (itself a parameterised type), and the result type — that is, types of kind `(* -> *) -> * -> *`. For example, the partiality monad transformer is represented by the type

```
newtype MaybeT m a = MaybeT (m (Maybe a))
```

According to the definition, `MaybeT m` should be a monad whenever `m` is, which we can demonstrate by implementing `return` and `>>=`:

```
instance Monad m => Monad (MaybeT m) where
  return x = MaybeT (return (Just x))
  MaybeT m >>= f =
    MaybeT (do x <- m
               case x of
                 Nothing -> return Nothing
                 Just a -> let MaybeT m' = f a in m')
```

Moreover, according to the definition of a monad transformer above, there should also be a monad morphism from `m` to `MaybeT m` — that is, it should be possible to transform computations of one type into the other. Since we need to define monad morphisms for many different monad transformers, we use Haskell's overloading again and introduce a *class of monad transformers*

```
class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a
```

Here `t` is the monad transformer, `m` is the monad it is applied to, and `lift` is the monad morphism[7]. Now we can make `MaybeT` into an instance of this class:

```
instance Monad m => MonadTransformer MaybeT m where
  lift m = MaybeT (do x <- m
                      return (Just x))
```

The purpose of the `MaybeT` transformer is to enable computations to fail: we shall introduce operations to cause and handle failures. One might expect their types to be

```
failure :: MaybeT m a
handle :: MaybeT m a -> MaybeT m a -> MaybeT m a
```

However, this is not good enough: since we expect to combine `MaybeT` with other monad transformers, the monad we actually want to apply these operations at may well be of some other form — but as long as it involves the `MaybeT` transformer somewhere, we ought to be able to do so.

We will therefore overload these operations also, and define a class of '`Maybe`-like' monads[8]:

```
class Monad m => MaybeMonad m where
  failure :: m a
  handle :: m a -> m a -> m a
```

Of course, monads of the form `MaybeT m` will be instances of this class, but later we will also see others. In this case, the instance declaration is

```
instance Monad m => MaybeMonad (MaybeT m) where
  failure = MaybeT (return Nothing)
  MaybeT m `handle` MaybeT m' =
    MaybeT (do x <- m
               case x of
                 Nothing -> m'
                 Just a -> return (Just a))
```

[7] Here we step outside Haskell 98 by using a *multiple parameter class* – an extension which is, however, supported by Hugs and many other implementations. We make m a parameter of the class to permit the definition of monad transformers which place additional requirements on their argument monad.

[8] Usually the standard Haskell class MonadPlus with operations mzero and mplus is used in this case, but in the present context the names MaybeMonad, failure and handle are more natural.

Finally, we need a way to 'run' elements of this type. We define

```
runMaybe :: Monad m => MaybeT m a -> m a
runMaybe (MaybeT m) = do x <- m
                         case x of
                           Just a -> return a
```

for this purpose. (We leave undefined how we 'run' an erroneous computation, thus converting an explicitly represented error into a real Haskell one).

We have now seen all the elements of a monad transformer in Haskell. To summarise:

- We define a type to represent the transformer, say `TransT`, with two parameters, the first of which should be a monad.
- We declare `TransT m` to be a `Monad`, under the assumption that `m` already is.
- We declare `TransT` to be an instance of class `MonadTransformer`, thus defining how computations are lifted from `m` to `TransT m`.
- We define a class `TransMonad` of '`Trans`-like monads', containing the operations that it is `TransT`'s purpose to support.
- We declare `TransT m` to be an instance of `TransMonad`, thus showing that it does indeed support them.
- We define a function to 'run' `(TransT m)`-computations, which produces `m`-computations as a result. In general `runTrans` may need additional parameters — for example, for a state transformer we probably want to supply an initial state.

We can carry out this program to define monad transformers for, among others,

- **state transformers**, represented by

  ```
  newtype StateT s m a = StateT (s -> m (s, a))
  ```

  supporting operations in the class[9]

  ```
  class Monad m => StateMonad s m | m -> s where
    readState :: m s
    writeState :: s -> m ()
  ```

- **environment readers**, represented by

  ```
  newtype EnvT s m a = EnvT (s -> m a)
  ```

  supporting operations in the class

[9] This class declaration uses Mark Jones' *functional dependencies*, supported by Hugs, to declare that the type of the monad's state is determined by the type of the monad itself. In other words, the same monad cannot have two different states of different types. While not strictly necessary, making the dependency explicit enables the type-checker to infer the type of the state much more often, and helps to avoid hard-to-understand error messages about ambiguous typings.

```
class Monad m => EnvMonad env m | m -> env where
  inEnv :: env -> m a -> m a
  rdEnv :: m env
```

where `rdEnv` reads the current value of the environment, and `inEnv` runs its argument in the given environment.

– **continuations**, represented by

```
newtype ContT ans m a = ContT ((a -> m ans) -> m ans)
```

supporting operations in the class

```
class Monad m => ContMonad m where
  callcc :: ((a -> m b) -> m a) -> m a
```

where `callcc f` calls `f`, passing it a function `k`, which if it is ever called terminates the call of `callcc` immediately, with its argument as the final result.

Two steps remain before we can use monad transformers in practice. Firstly, since monad transformers only transform one monad into another, we must define a monad to start with. Although one could start with any monad, it is natural to use a 'vanilla' monad with no computational features – the *identity* monad

```
newtype Id a = Id a
```

The implementations of `return` and `>>=` on this monad just add and remove the `Id` tag.

Secondly, so far the *only* instances in class `MaybeMonad` are of the form `MaybeT m`, the only instances in class `StateMonad` of the form `StateT s m`, and so on. Yet when we combine two or more monads, of course we expect to use the features of *both* in the resulting monad. For example, if we construct the monad `StateT s (MaybeT Id)`, then we expect to be able to use `failure` and `handle` at this type, as well as `readState` and `writeState`.

The only way to do so is to give further instance declarations, which define how to 'lift' the operations of one monad over another. For example, we can lift failure handling to state monads as follows:

```
instance MaybeMonad m => MaybeMonad (StateT s m) where
  failure = lift failure
  StateT m `handle` StateT m' = StateT (\s -> m s `handle` m' s)
```

Certainly this requires $O(n^2)$ instance declarations, one for each pair of monad transformers, but there is unfortunately no other solution.

The payoff for all this work is that, when we need to define a monad, we can often construct it quickly by composing monad transformers, and automatically inherit a collection of useful operations.

Example 46. We can implement the parsing library from section 5.3 by combining state transformation with failure. We shall let a parser's state be the input to be parsed; running a parser will consume a part of it, so running two parsers in sequence will parse successive parts of the input. Attempting to run a parser may succeed or fail, and we will often wish to handle failures by trying a different parser instead. We can therefore define a suitable monad by

```
type Parser a = StateT String (MaybeT Id) a
```

whose computations we can run using

```
runParser p s = runId (runMaybe (runState s p))
```

It turns out that the operator we called `|||` earlier is just `handle`, and `satisfy` is simply defined by

```
satisfy :: (s -> Bool) -> Parser s s
satisfy p = do s<-readState
               case s of
                 [] -> failure
                 x:xs -> if p x then do writeState xs
                                        return x
                                 else failure
```

There is no more to do.

6.1 Monads and DSLs: a Discussion

It is clear why monads have been so successful for programming I/O and imperative algorithms in Haskell — they offer the only really satisfactory solution. But they have also been widely adopted by the designers of combinator libraries. Why? We have made the analogy between a combinator library and a domain specific language, and since monads can be used to structure denotational semantics, it is not so surprising that they can also be used in combinator libraries. But that something *can* be used, does not mean that it *will* be used. The designer of a combinator library has a choice: he need not slavishly follow the One Monadic Path — why, then, have so many chosen to do so? What are the overwhelming practical benefits that flow from using monads in combinator library design in particular?

Monads offer significant advantages in three key areas. Firstly, they offer a *design principle* to follow. A designer who wants to capture a particular functionality in a library, but is unsure exactly what interface to provide to it, can be reasonably confident that a monadic interface will be a good choice. The monad interface has been tried and tested: we know it allows the library user great flexibility. In contrast, early parsing libraries, for example, used non-monadic interfaces which made some parsers awkward to write.

Secondly, monads can *guide the implementation* of a library. A library designer must choose an appropriate type for his combinators to work over, and his

task is eased if the type is a monad. Many monad types can be constructed systematically, as we have seen in this section, and so can some parts of the library which operate on them. Given a collection of monad transformers, substantial parts of the library come 'for free', just as when we found there was little left to implement after composing the representation of `Parser`s from two monad transformers.

Thirdly, there are benefits when many libraries *share a part of their interfaces*. Users can learn to use each new library more quickly, because the monadic part of its interface is already familiar. Because of the common interface, it is reasonable to define generic monadic functions, such as `liftM2`, which work with any monadic library. This both helps users, who need only learn to use `liftM2` once, and greatly eases the task of implementors, who find much of the functionality they want to provide comes for free. And of course, it is thanks to the widespread use of monads that Haskell has been extended with syntactic sugar to support them — if each library had its own completely separate interface, then it would be impractical to support them all with special syntax.

Taken together, these are compelling reasons for a library designer to choose monads whenever possible.

7 Exercises on Monads

This section contains practical exercises, intended to be solved using Hugs on a computer. Since some readers will already be familiar with Haskell and will have used monads already, while others will be seeing them for the first time, the exercises are divided into different levels of difficulty. Choose those which are right for you.

The Hugs interpreter is started with the command

```
hugs -98
```

The flag informs `hugs` that extensions to Haskell 98 should be allowed — and they are needed for some of these exercises. When Hugs is started it prompts for a command or an expression to evaluate; the command ":?" lists the commands available. Hugs is used by placing definitions in a file, loading the file into the interpreter (with the ":l" or ":r" command), and typing expressions to evaluate. You can obtain information on any defined name with the command ":i", and discover which names are in scope using ":n" followed by a regular expression matching the names you are interested in. Do not try to type definitions in response to the interpreter's prompt: they will not be understood.

7.1 Easy Exercises

Choose these exercises if you were previously unfamiliar with monads or Haskell.

Exercise 47. Write a function

```
dir :: IO [String]
```

which returns a list of the file names in the current directory. You can obtain them by running `ls` and placing the output in a file, which you then read. You will need to import module `System`, which defines a function `system` to execute shell commands — place "`import System`" on the first line of your file. A string can be split into its constituent words using the standard function `words`, and you can print values (for testing) using the standard function `print`.

Exercise 48. Write a function

```
nodups :: [String] -> [String]
```

which removes duplicate elements from a list of strings — the intention is to return a list of strings in the argument, in order of first occurrence. It is easy to write an inefficient version of nodups, which keeps a list of the strings seen so far, but you should use a hash table internally so that each string in the input is compared against only a few others. (The choice of hash function is not particularly important for this exercise, though). Moreover, you should produce the result list *lazily*. Test this by running

```
interact (unlines . nodups . lines)
```

which should echo each line you then type on its first occurrence.

You will need to use Haskell lists, which are written by enclosing their elements in square brackets separated by commas, and the cons operator, which is ":". Import module `LazyST`, and use `newSTArray` to create your hash table, `readSTArray` to read it, and `writeSTArray` to write it. Beware of Haskell's layout rule, which insists that every expression in a `do` begin in the same column — and interprets everything appearing in that column as the start of a new expression.

Exercise 49. The implementation of the `MaybeT` transformer is given above, but the implementations of the `StateT`, `EnvT` and `ContT` transformers were only sketched. Complete them. (`ContT` is quite difficult, and you might want to leave it for later).

Exercise 50. We define the `MaybeT` type by

```
newtype MaybeT m a = MaybeT (m (Maybe a))
```

What if we had defined it by

```
newtype MaybeT m a = MaybeT (Maybe (m a))
```

instead? Could we still have defined a monad transformer based on it?

Exercise 51. We defined the type of `Parser`s above by

```
type Parser a = StateT String (MaybeT Id) a
```

What if we had combined state transformation and failure the other way round?

```
type Parser a = MaybeT (StateT String Id) a
```

Define an instance of `StateMonad` for `MaybeT`, and investigate the behaviour of several examples combining failure handling and side-effects using each of these two monads. Is there a difference in their behaviour?

7.2 Moderate Exercises

Choose these exercises if you are comfortable with Haskell, and have seen monads before.

Exercise 52. Implement a monad `MaybeST` based on the built-in `ST` monad, which provides updateable typed references, but also supports failure and failure handling. If `m` fails in `m `handle` h`, then all references should contain the *same* values on entering the handler `h` that they had when `m` was entered.

Can you add an operator

```
commit :: MaybeST ()
```

with the property that updates before a `commit` survive a subsequent failure?

Exercise 53. A different way to handle failures is using the type

```
newtype CPSMaybe ans a =
  CPSMaybe ((a -> ans -> ans) -> ans -> ans)
```

This is similar to the monad of continuations, but both computations and continuations take an extra argument — the value to return in case of failure. When a failure occurs, this argument is returned directly and the normal continuation is not invoked.

Make `CPSMaybe` an instance of class `Monad` and `MaybeMonad`, and define `runCPSMaybe`.

Failure handling programs often use a great deal of space, because failure handlers retain data that is no longer needed in the successful execution. Yet once one branch has progressed sufficiently far, we often know that its failure handler is no longer relevant. For example, in parsers we usually combine parsers for quite different constructions, and if the first parser succeeds in parsing more than a few tokens, then we know that the second cannot possibly succeed. Can you define an operator

```
cut :: CPSMaybe ans ()
```

which discards the failure handler, so that the memory it occupies can be reclaimed? How would you use `cut` in a parsing library?

7.3 Difficult Exercises

These should give you something to get your teeth into!

Exercise 54. Implement a domain specific language for concurrent programming, using a monad `Process s a` and typed channels `Chan s a`, with the operations

```
chan :: Process s (Chan s a)
send :: Chan s a -> a -> Process s ()
recv :: Chan s a -> Process s a
```

to create channels and send and receive messages (synchronously),

```
fork :: Process s a -> Process s ()
```

to start a new concurrent task, and

```
runProcess :: (forall s. Process s a) -> a
```

to run a process. By analogy with the `ST` monad, `s` is a state-thread identifier which is used to guarantee that channels are not created in one call of `runProcess` and used in another. You will need to write the type of `runProcess` *explicitly* — Hugs cannot infer rank 2 types.

Exercise 55. Prolog provides so-called *logical variables*, whose values can be referred to before they are set. Define a type `LVar` and a monad `Logic` in terms of `ST`, supporting operations

```
newLVar :: Logic s (LVar s a)
readLVar :: LVar s a -> a
writeLVar :: LVar s a -> a -> Logic s ()
```

where `s` is again a state-thread identifier. The intention is that an `LVar` should be written exactly once, but its value may be read *beforehand*, between its creation and the write — lazy evaluation is at work here. Note that `readLVar` does *not* have a monadic type, and so can be used anywhere. Of course, this can only work if the value written to the `LVar` does not depend on itself. *Hint:* You will need to use

```
fixST :: (a -> ST s a) -> ST s a
```

to solve this exercise — `fixST (\x -> m)` binds `x` to the result produced by `m` during its own computation.

Exercise 56. In some applications it is useful to dump the state of a program to a file, or send it over a network, so that the program can be restarted in the same state later or on another machine. Define a monad `Interruptable`, with an operation

```
dump :: Interruptable ()
```

which stops execution and converts a representation of the state of the program to a form that can be saved in a file. The result of running an `Interruptable` computation should indicate whether or not dumping occurred, and if so, provide the dumped state. If `s` is a state dumped by a computation `m`, then `resume m s` should restart `m` in the state that `s` represents. Note that `m` might dump several times during its execution, and you should be able restart it at each point.

You will need to choose a representation for states that can include every type of value used in a computation. To avoid typing problems, convert values to strings for storage using `show`.

You will not be able to make `Interruptable` an instance of class `Monad`, because your implementations of `return` and `>>=` will not be sufficiently polymorphic — they will only work over values that can be converted to strings. This is unfortunate, but you can just choose other names for the purposes of this exercise. One solution to the problem is described in [Hug99].

8 Intermediate Languages for Compilation

We have seen how monads may be used to structure the denotational semantics of languages with implicit computational effects and how they may be used to express and control the use of computational effects in languages like Haskell, in which the only implicit effect is the possibility of non-termination. We now turn to the use of monads in the practical compilation of languages, such as ML, with implicit side effects. Much of this material refers to the MLj compiler for Standard ML [BKR98], and its intermediate language MIL (Monadic Intermediate Language) [BK99].

8.1 Compilation by Transformation

It should not be a surprise that ideas which are useful in structuring semantics also turn out to be useful in structuring the internals of compilers since, by implementing rules for deriving program equivalences, there is a sense in which compilers actually *do* semantics. Even in the absence of sophisticated static analyses, compilers for functional languages typically work by translating the user's program into an intermediate form and then performing a sequence of rewrites on the intermediate representation before translating that into lower-level code in the backend. These rewrites are intended to preserve the semantics (i.e. the observable behaviour) of the user's program, whilst improving the efficiency of the final program in terms of execution speed, dynamic memory usage and/or code size. Hence the rewriting rules used by the compiler should be observational equivalences, and if they are applied locally (i.e. independently of the surrounding context) then they should be instances of an observational *congruence* relation. Of course, the hard part is that the compiler also has to decide when applying a particular semantic equation is likely to be an improvement.

8.2 Intermediate Languages

The reasons for having an intermediate language at all, rather than just doing rewriting on the abstract syntax tree of the source program, include:

1. Complexity. Source languages tend to have many sophisticated syntactic forms (e.g. nested patterns or list comprehensions) which are convenient for the programmer but which can be translated into a simpler core language, leaving fewer cases for the optimizer and code generator to deal with.
2. Level. Many optimizing transformations involve choices which cannot be expressed in the source language because they are at a lower level of abstraction. In other words, they involve distinctions between implementation details which the source language cannot make. For example
   - All functions in ML take a single argument – if you want to pass more than one then you package them up as a single tuple. This is simple and elegant for the programmer, but we don't want the compiled code to pass a pointer to a fresh heap-allocated tuple if it could just pass

a couple of arguments on the stack or in registers. Hence MIL (like other intermediate languages for ML) includes both tuples and multiple arguments and transforms some instances of the former into the latter.
- MIL also includes datastructures with 'holes' (i.e. uninitialized values). These are used to express a transformation which turns some non-tail calls into tail calls and have linear typing rules which prevent holes being dereferenced or filled more than once [Min98].

Of course, there are many levels of abstraction between the source and target languages, so it is common for compilers to use several different intermediate languages at different phases.[10]

However, in these notes we shall not be concerned so much with the complexity of realistic source languages, or with expressing low-level implementation details in intermediate languages. Instead, we will be interested in the slightly more nebulous idea that a good intermediate language may be more *uniform*, *expressive* and *explicit* than the source.

Many important transformations do not involve concepts which are essentially at a lower-level level of abstraction than the source language, but can nevertheless be anywhere between bothersome and impossible to express or implement directly on the source language syntax.

The equational theory of even a simplified core of the source language may be messy and ill-suited to optimization by rewriting. Rather than have a complex rewriting system with conditional rewrites depending on various kinds of contextual information, one can often achieve the same end result by translating into an intermediate language with a better-behaved equational theory. It is typically the case that a 'cleaner' intermediate language makes explicit some aspects of behaviour which are implicit in the source language.[11] Examples:

- Many intermediate languages introduce explicit names for every intermediate value. Not only are the names useful in building various auxiliary datastructures, but they make it easy to, for example, share subexpressions. A very trivial case would be

  ```
  let val x = ((3,4),5)
  in (#1 x, #1 x)
  end
  ```

  which we *don't* want to simplify to the equivalent

  ```
  ((3,4),(3,4))
  ```

  because that allocates two identical pairs. One particularly straighforward way to get a better result is to only allow introductions and eliminations to be applied to variables or atomic constants, so the translation of the original program into the intermediate form is

[10] Or to have one all-encompassing intermediate datatype, but the ensure that the input and output of each phase satisfy particular additional constraints.

[11] Which can make such intermediate representations larger than the corresponding source.

```
let val y = (3,4)
in let val x = (y,5)
   in (#1 x, #1 x)
   end
end
```

which rewrites to

```
let val y = (3,4)
in let val x = (y,5)
   in (y, y)
   end
end
```

and then to

```
let val y = (3,4)
in (y, y)
end
```

which is probably what we wanted.

- MIL contains an unusual exception-handling construct because SML's `handle` construct is unable to express some commuting conversion-style rewrites which we wished to perform [BK01].
- Some compilers for higher-order languages use a continuation passing style (CPS) lambda-calculus as their intermediate language (see, for example, [App92,KKR+86]). There are translations of call by value (CBV) and call by name (CBN) source languages into CPS. Once a program is in CPS, it is sound to apply the full unrestricted β, η rules, rather than, say, the more restricted β_v, η_v rules which are valid for λ_v (the CBV lambda calculus). Moreover, as Plotkin shows in his seminal paper [Plo75], β and η on CPS terms prove strictly more equivalences between translated terms than do β_v and η_v on the corresponding λ_v terms. Hence, a compiler for a CBV language which translates into CPS and uses $\beta\eta$ can perform more transformations than one which just uses β_v and η_v on the source syntax.
  CPS transformed terms make evaluation order explict (which makes them easier to compile to low-level imperative code in the backend), allow tail-call elimination be be expressed naturally, and are particularly natural if the language contains `call/cc` or other sophisticated control operators.
  However, Flanagan et al. [FSDF93] argue that compiling CBV lambda-calculus via CPS is an unnecessarily complicated and indirect technique. The translation introduces lots of new λ-abstractions and new, essentially trivial, 'administrative redexes'. To generate good code, and to identify administrative redexes, real CPS compilers treat abstractions introduced by the translation process differently from those originating in the original program and effectively undo the CPS translation in the backend, after having performed transformations. Flanagan et al. show that the same effect can be obtained by using a λ-calculus with `let` and peforming *A-reductions* to reach

an *A-normal form*. A-reductions were introduced in [SF93] and are defined in terms of *evaluation contexts*. Amongst other things, A-normal forms name all intermediate values and only apply eliminations to variables or values. An example of an A-reduction is the following:

$$\mathcal{E}[\text{if } V \text{ then } N_1 \text{ else } N_2] \longrightarrow \text{if } V \text{ then } \mathcal{E}[N_1] \text{ else } \mathcal{E}[N_2]$$

where $\mathcal{E}[\cdot]$ is an evaluation context. Flanagan et al. observe that most non-CPS ('direct style') compilers perform some A-reductions in a more-or-less ad hoc manner, and suggest that doing all of them, and so working with A-normal forms, is both more uniform and leads to faster code.

Typed Intermediate Languages One big decision when designing an intermediate language is whether or not it should be typed. Even when the source language has strong static types, many compilers discard all types after they have been checked, and work with an untyped intermediate language. More recently, typed intermediate languages have become much more popular in compilers (and in the fashionable area of mobile code security). Examples of typed compiler intermediate languages include FLINT [Sha97], the GHC intermediate language [Jon96] and MIL. The advantages of keeping type information around in an intermediate language include:

- Types are increasingly the basis for static analyses, optimizing transformations and representation choices. Type-based optimization can range from the use of sophisticated type systems for static analyses to exploitation of the fact that static types in the source language provide valuable information which it would be foolish to ignore or recompute. For example, the fact that in many languages pointers to objects of different types can never alias can be used to allow more transformations. The MLj compiler uses simple type information to share representations, using a single Java class to implement several different ML closures.
- Type information can be used in generating backend code, for example in interfacing to a garbage collector or allocating registers.
- Type-checking the intermediate representation is a very good way of finding compiler bugs.[12]
- It's particularly natural if the language allows types to be reflected as values.
- It's clearly the right thing to do if the target language is itself typed. This is the case for MLj (since Java bytecode is typed) and for compilers targetting typed assembly language [MWCG99].

But there are disadvantages too:

- Keeping type information around and maintaining it during transformations can be very expensive in both space and time.

[12] And this really *is* a significant advantage!

– Unless the type system is complex and/or rather non-standard, restricting the compiler to work with typable terms can prohibit transformations. Even something like closure-conversion (packaging functions with the values of their free variables) is not trivial from the point of view of typing [MMH96].

λML$_T$ as a Compiler Intermediate Language Several researchers have suggested that Moggi's computational metalanguage $\lambda\mathrm{ML}_T$ [Mog89,Mog91] might be useful as the basis of a typed intermediate language.[13]

Benton [Ben92] proposed the use of the computational metalanguage as a way of expressing the optimizations which may be performed as a result of strictness analysis in compilers for CBN languages such as Haskell. Earlier work on expressing the use of strictness analysis was largely in terms of a somewhat informal notion of changes in 'evaluation strategy' for fixed syntax. It is much more elegant to reason about changes in *translation* of the source language into some other language which itself has a fixed operational semantics. In the case of a pure CBN source language (such as PCF [Plo77]), however, one cannot (directly) use a source-to-source translation to express strictness-based transformations. Adding a strict `let` construct with typing rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N : B}$$

where let $x = M$ in N first evaluates M to Weak Head Normal Form (WHNF) before substituting for x in N, allows one to express basic strictness optimizations, such as replacing the application $M\ N$ with let $x = N$ in $(M\ x)$ when M is known to be strict. But this is only half the story – we'd also like to be able to perform optimizations based on the fact that certain expressions (such as x in our example) are known to be bound to values in WHNF and so need not be represented by thunks or re-evaluated. To capture this kind of information, Benton suggested a variant of the computational metalanguage in which an expression of a *value* type A is always in WHNF and the *computation* type TA is used for potentially unevaluated expressions which, if they terminate, will yield values of type A. The default translation of a call-by-name expression of type $A \to B$ is then to an intermediate language expression of type of type $T((A \to B)^n) = T(TA^n \to TB^n)$, i.e. a computation producing a function from computations to computations. An expression denoting a strict function which is only called in strict contexts, by contrast, can be translated into an intermediate language term of type $T(A^n \to TB^n)$: a computation producing a function from *values* to computations.

Exercise 57. The 'standard' denotational semantics of PCF is in the CCC of pointed ω-cpos and continuous maps, with $[\![\texttt{int}]\!] = \mathbb{Z}_\perp$ and function space interpreted by $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$. This semantics is adequate for a CBN operational

[13] The author, like several others, persistently refers to $\lambda\mathrm{ML}_T$ as the *computational lambda calculus*, although Moggi actually invented that name for his λ_c, a different calculus (without computational types). This doesn't seem to have ever caused terrible confusion, but one should be aware of it.

semantics in which the notion of observation is termination of closed terms of ground type. It seems natural that one could give a semantics to PCF with a strict let construct just by defining

$$[\![\text{let } x = M \text{ in } N]\!]\rho = \begin{cases} \bot & \text{if } [\![M]\!]\rho = \bot \\ [\![N]\!]\rho[x \mapsto [\![M]\!]\rho] & \text{otherwise} \end{cases}$$

but in fact, the semantics is then no longer adequate. Why? How might one modify the semantics to fix the problem? How good is the modified semantics as a semantics of the original language (i.e. without `let`)?

Other authors addressed the problem of expressing strictness-based transformations by varying a translation of the source language into continuation passing style [BM92,DH93]. The two strands of work were then brought together by Danvy and Hatcliff [HD94], who showed how various CPS transforms could be factored through translations into the computational metalanguage and how the administrative reductions of CPS, and Flanagan et al.'s A-reductions, corresponded to applying the β-reduction and *commuting conversions* (see Section 11) associated with the computation type constructor in the computational metalanguage. Danvy and Hatcliff also suggest that the computational metalanguage could make an attractive compiler intermediate language.

Peyton Jones et al. [JLST98] proposed the use of an intermediate language based on the computational metalanguage as a common framework for compiling both call-by-value and call-by-name languages.[14] Barthe et al. [BHT98] add computational types to the pure type systems (PTS) to obtain monadic versions of a whole family of higher-order typed lambda calculi (such as F_ω and the Calculus of Constructions) and advocate the use of such calculi as compiler intermediate languages for languages which combine polymorphic type and/or module systems with side-effects.

9 Type and Effect Systems

9.1 Introduction

The work referred to in the previous section concerns using a well-behaved intermediate language (A-normal forms, CPS or λML_T) to perform sound rewriting on a programs written in languages with 'impure' features. All those intermediate languages make some kind of separation (in the type system and/or the language syntax) between 'pure' values and 'impure' (potentially side-effecting) computations. The separation is, however, fairly crude and there are often good reasons for wanting to infer at compile-time a safe approximation[15] to just *which*

[14] Unfortunately, [JLST98] contains an error: the semantics of the intermediate language $\mathcal{L}_2$ does not actually satisfy the monad equations.

[15] As is always the case with static analyses, precise information is uncomputable, so we have to settle for approximations. In this case, that means overestimating the possible side-effects of an expression.

side-effects may happen as a result of evaluating a particular expression. This kind of *effect analysis* is really only applicable to CBV languages, since CBN languages do not usually allow any side-effects other than non-termination.

Historically, the first effect analyses for higher order languages were developed to avoid a type soundness problem which occurs when polymorphism is combined naively with updateable references. To see the problem, consider the following (illegal) SML program:

```
let val r = ref (fn x=> x)
in (r := (fn n=>n+1);
    !r true
   )
end
```

Using the 'obvious' extension of the Hindley-Milner type inference rules to cover reference creation, dereferencing and assignment, the program above would typecheck:

1. `(fn x=>x)` has type $\alpha \to \alpha$, so
2. `ref (fn x=>x)` has type $(\alpha \to \alpha)$`ref`
3. generalization then gives `r` the type scheme $\forall\alpha.(\alpha \to \alpha)$`ref`
4. so by specialization `r` has type (`int` $\to$ `int`)`ref`, meaning the assignment typechecks
5. and by another specialization, `r` has type (`bool` $\to$ `bool`)`ref`, so
6. `!r` has type `bool` $\to$ `bool`, so the application type checks.

However, it is clear that the program really has a type error, as it will try to increment a boolean.

To get around this problem, Gifford, Lucassen, Jouvelot, Talpin and others [GL86,GJLS87,TJ94] developed *type and effect systems*. The idea is to have a refined type system which infers both the type *and* the possible effects which an expression may have, and to restrict polymorphic generalization to type variables which do not appear in side-effecting expressions. In the example above, one would then infer that the expression `ref (fn x=>x)` creates a new reference cell of type $\alpha \to \alpha$. This prevents the type of `r` being generalized in the `let` rule, so the assignment causes α to be unified with `int` and the application of `!r` to `true` then fails to typecheck.[16]

It should be noted in passing that there are a number of different ways of avoiding the type loophole. For example, Tofte's imperative type discipline [Tof87] using 'imperative type variables' was used in the old (1990) version of the Standard ML Definition, whilst Leroy and Weis proposed a different scheme for tracking 'dangerous' type variables (those appearing free in the types of expressions stored in references) [LW91]. A key motivation for most of that work was

[16] Depending on the order in which the inference algorithm works, the application might alternatively cause α to be unified with `bool` and then the error would be discovered in the assignment. This is an example of why giving good type error messages is hard.

to allow as much polymorphic generalization as possible to happen in the `let` rule, whilst still keeping the type system sound. However, expensive and unpredictable inference systems which have a direct impact on which user programs actually typecheck are not often a good idea. In 1995, Wright published a study [Wri95] indicating that nearly all existing SML code would still typecheck and run identically (sometimes modulo a little η-expansion) if polymorphic generalization were simply restricted to source expressions which were syntactic values (and thus trivially side-effect free). This simple restriction was adopted in the revised (1997) SML Definition and research into fancy type systems for impure polymorphic languages seems to have now essentially ceased.

However, there are still very good reasons for wanting to do automatic effect inference. The most obvious is that more detailed effect information allows compilers to perform more aggressive optimizations. Other applications include various kinds of verification tool, either to assist the programmer or to check security policies, for example. In SML, even a seemingly trivial rewrite, such as the dead-code elimination

$$\texttt{let val x =}\ M_1\ \texttt{in}\ M_2\ \texttt{end} \quad \longrightarrow \quad M_2 \qquad (\texttt{x} \notin FV(M_2))$$

is generally only valid if the evaluation of M_1 doesn't diverge, perform I/O, update the state or throw an exception (though it is still valid if M_1 reads from reference cells or allocates new ones).[17]

9.2 The Basic Idea

There are now many different type and effect systems in the literature, but they all share a common core. (The book [NHH99] contains, amongst other things, a fair amount on effect systems and many more references than these notes.) A traditional type system infers judgements of the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : B$$

where the A_i and B are types. A type and effect system infers judgements of the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : B, \varepsilon$$

which says that in the given typing context, the expression M has type B and effect ε. The effect ε is drawn from some set $\mathcal{E}$ whose elements denote *sets* of actual effects which may occur at runtime (in other words, they are *abstractions* of runtime effects, just as types are abstractions of runtime values). Exactly what is in $\mathcal{E}$ depends not only on what runtime effects are possible in the language, but also on how precise one wishes to make the analysis. The simplest non-trivial effect system would simply take $\mathcal{E}$ to have two elements, one (usually written $\emptyset$)

[17] The reader who thinks that this is a silly example because 'programmers never write code like that' is quite mistaken. Immediately unused bindings may not be common in the original source, but they are frequently created as a result of other rewrites.

denoting no effect at all ('pure'), and the other just meaning 'possibly has some effect'. Most effect systems are, as we shall see, a little more refined than this.

The first thing to remark about the form of a type and effect judgement is that an effect appears on the right of the turnstile, but not on the left. This is because we are only considering CBV languages, and that means that at runtime free variables will always be bound to *values*, which have no effect. An effect system for an impure CBN language, were there any such thing, would have pairs of types and effects in the context too.[18] Because variables are always bound to values, the associated type and effect rule will be:

$$\frac{}{\Gamma, x : A \vdash x : A, \emptyset}$$

The second point is that $\mathcal{E}$ actually needs to be an algebra, rather than just a set; i.e. it has some operations for combining effects defined on it. Consider the effectful version of the rule for a simple (strict, non-polymorphic, non-computational) `let` expression:

$$\frac{\Gamma \vdash M : A, \varepsilon_1 \qquad \Gamma, x : A \vdash N : B, \varepsilon_2}{\Gamma \vdash \text{let } x = M \text{ in } N : B, ?}$$

What should the effect of the compound expression be? Dynamically, M will be evaluated, possibly performing some side-effect from the set denoted by ε_1 and, assuming the evaluation of M terminated with a value V, then $N[V/x]$ will be evaluated and possibly perform some side-effect from the set denoted by ε_2. How we combine ε_1 and ε_2 depends on how much accuracy we are willing to pay for in our static analysis. If we care about the relative ordering of side-effects then we might take elements of $\mathcal{E}$ to denote sets of *sequences* (e.g. regular languages) over some basic set of effects and then use language concatenation $\varepsilon_1 \cdot \varepsilon_2$ to combine the effects in the `let` rule. Commonly, however, we abstract away from the relative sequencing and multiplicity of effects and just consider *sets* of basic effects. In this case the natural combining operation for the `let` rule is some abstract union operation.[19]

For the conditional expression, the following is a natural rule:

$$\frac{\Gamma \vdash M : \text{bool}, \varepsilon' \qquad \Gamma \vdash N_1 : A, \varepsilon_1 \qquad \Gamma \vdash N_2 : A, \varepsilon_2}{\Gamma \vdash (\text{if } M \text{ then } N_1 \text{ else } N_2) : A, \varepsilon' \cdot (\varepsilon_1 \cup \varepsilon_2)}$$

If we were not tracking sequencing or multiplicity, then the effect in the conclusion of the `if` rule would just be $\varepsilon' \cup \varepsilon_1 \cup \varepsilon_2$, of course.

[18] Although the mixture of CBN and side-effects is an unpredictable one, Haskell does actually allow it, via the 'experts-only' `unsafePerformIO` operation. But I'm still not aware of any type and effect system for a CBN language.

[19] Effect systems in the literature often include a binary $\cup$ operation in the formal syntax of effect annotations, which are then considered modulo unit, associativity, commutativity and idempotence. For very simple effect systems, this is unnecessarily syntactic, but it's not so easy to avoid when one also has effect variables and substitution.

The other main interesting feature of almost all type and effect systems is the form of the rules for abstraction and application, which make types dependent on effects, in that the function space constructor is now annotated with a 'latent effect' $A \xrightarrow{\varepsilon} B$. The rule for abstraction looks like:

$$\frac{\Gamma, x : A \vdash M : B, \varepsilon}{\Gamma \vdash (\lambda x : A.\ M) : A \xrightarrow{\varepsilon} B, \emptyset}$$

because the λ-abstraction itself is a value, and so has no immediate effect ($\emptyset$) but will have effect ε when it is applied, as can be seen in the rule for application:

$$\frac{\Gamma \vdash M : A \xrightarrow{\varepsilon_1} B, \varepsilon_2 \qquad \Gamma \vdash N : A, \varepsilon_3}{\Gamma \vdash M\ N : B, \varepsilon_2 \cdot \varepsilon_3 \cdot \varepsilon_1}$$

The overall effect of evaluating the application is made up of three separate effects – that which occurs when the function is evaluated, that which occurs when the argument is evaluated and finally that which occurs when the body of the function is evaluated. (Again, most effect systems work with sets rather than sequences, so the combining operation in the conclusion of the application rule is just $\cup$.)

The final thing we need to add to our minimal skeleton effect system is some way to weaken effects. The collection $\mathcal{E}$ of effects for a given analysis always has a natural partial order relation $\subseteq$ defined on it such that $\varepsilon \subseteq \varepsilon'$ means ε' denotes a larger set of possible runtime side-effects than ε. Typically $\subseteq$ is just the subset relation on sets of primitive effects. The simplest rule we can add to make a usable system is the *subeffecting* rule:

$$\frac{\Gamma \vdash M : A, \varepsilon \qquad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash M : A, \varepsilon'}$$

Exercise 58. Define a toy simply-typed CBV functional language (integers, booleans, pairs, functions, recursion) with a *fixed* collection of global, mutable integer variables. Give it an operational and/or denotational semantics. Give a type and effect system (with subeffecting) for your language which tracks which global variables may be read and written during the evaluation of each expression (so an effect will be a pair of sets of global variable names). Formulate and prove a soundness result for your analysis. Are there any closed terms in your language which require the use of the subeffect rule to be typable at all?

9.3 More Precise Effect Systems

One of the great things about static analyses is that one can always tweak any analysis system to make it more accurate.[20] There are a number of natural and popular ways to improve the precision of the hopelessly weak 'simple-types' approach to effect analysis sketched in the previous section.

[20] This corollary to the unsolvability of the Halting Problem is known as 'The Full Employment Theorem for Compiler Writers'.

Subtyping The bidirectional flow of information in type systems or analyses which simply constrain types to be equal frequently leads to an undesirable loss of precision. For example, consider an effect analysis of the following very silly ML program (and forget polymorphism for the moment):

```
let fun f x = ()
    fun pure () = ()
    fun impure () = print "I'm a side-effect"
    val m = (f pure, f impure)
in pure
end
```

If they were typed in isolation, the best type for `pure` would be $\texttt{unit} \xrightarrow{\emptyset} \texttt{unit}$ and `impure` would get $\texttt{unit} \xrightarrow{\{\text{print}\}} \texttt{unit}$ (assuming that the constant `print` has type $\texttt{string} \xrightarrow{\{print\}} \texttt{unit}$). However, the fact that both of them get passed to the function `f` means that we end up having to make their types, including the latent effects, identical. This we can do by applying the subeffecting rule to the body of `pure` and hence deriving the same type $\texttt{unit} \xrightarrow{\{\text{print}\}} \texttt{unit}$ for both `pure` and `impure`. But then that ends up being the type inferred for the whole expression, when it's blindingly obvious that we should have been able to deduce the more accurate type $\texttt{unit} \xrightarrow{\emptyset} \texttt{unit}$.

The fact that the argument type of `f` has to be an impure function type has propogated all the way back to the *definition* of `pure`. Peyton Jones has given this phenomenon the rather apt name of the *poisoning problem*. One solution is to extend to notion of subeffecting to allow more general *subtyping*. We replace the subeffecting rule with

$$\frac{\Gamma \vdash M : A, \varepsilon \qquad \varepsilon \subseteq \varepsilon' \qquad A \leq A'}{\Gamma \vdash M : A', \varepsilon'}$$

where $\leq$ is a partial order on types defined by rules like

$$\frac{A' \leq A \qquad B \leq B' \qquad \varepsilon \subseteq \varepsilon'}{A \xrightarrow{\varepsilon} B \;\leq\; A' \xrightarrow{\varepsilon'} B'} \qquad \text{and} \qquad \frac{A \leq A' \qquad B \leq B'}{A \times B \;\leq\; A' \times B'}$$

Note the *contravariance* of the function space constructor in the argument type.

Using the subtyping rule we can now get the type and effect we'd expect for our silly example. The definitions of `pure` and `impure` are given different types, but we can apply the subtyping rule (writing 1 for `unit`)

$$\frac{\Gamma, \text{pure} : (1 \xrightarrow{\emptyset} 1) \vdash \text{pure} : (1 \xrightarrow{\emptyset} 1), \emptyset \qquad \dfrac{1 \leq 1 \quad 1 \leq 1 \quad \emptyset \subseteq \{\text{print}\}}{(1 \xrightarrow{\emptyset} 1) \;\leq\; (1 \xrightarrow{\{\text{print}\}} 1)} \qquad \emptyset \subseteq \emptyset}{\Gamma, \text{pure} : (1 \xrightarrow{\emptyset} 1) \vdash \text{pure} : (1 \xrightarrow{\{\text{print}\}} 1), \emptyset}$$

to coerce the *use* of `pure` when it is passed to `f` to match the required argument type whilst still using the more accurate type inferred at the point of definition as the type of the whole expression.

Effect Polymorphism Another approach to the poisoning problem is to introduce ML-style polymorphism at the level of effects (this is largely orthogonal to whether we also have polymorphism at the level of types). We allow effects to contain effect variables and then to allow the context to bind identifiers to type schemes, which quantify over effect variables.

Consider the following program

```
let fun run f = f ()
    fun pure () = ()
    fun impure () =  print "Poison"
    fun h () = run impure
in run pure
end
```

In this case, even with subtyping, we end up deriving a type and effect of `unit`, {print} for the whole program, though it actually has no side effect. With effect polymorphism, we can express the fact that there is a dependency between the effect of a particular call to `run` and the latent effect of the function which is passed at that point. The definition of `run` gets the type scheme

$$\forall a.(\texttt{unit} \xrightarrow{a} \texttt{unit}) \xrightarrow{a} \texttt{unit}$$

which is instantiated with $a = \emptyset$ in the application to `pure` and $a = \{\text{print}\}$ in the application to `impure` (which is actually never executed). That lets us deduce a type and effect of $\texttt{unit}, \emptyset$ for the whole program.

Regions One of the most influential ideas to have come out of work on type and effect systems is that of *regions*: static abstractions for sets of dynamically allocated run-time locations. If (as in the earlier exercise) one is designing an effect system to track the use of mutable storage in a language with a *fixed* set of global locations, there are two obvious choices for how precisely one tracks the effects – either one records simply whether or not an expression might read or write some unspecified locations, or one records a set of just *which* locations might be read or written. Clearly the second is more precise and can be used to enable more transformations. For example, the evaluation of an expression whose only effect is to read some locations might be moved from after to before the evaluation of an expression whose effect is to write some locations *if* the set of locations possibly read is disjoint from the set of locations possibly written.

But no real programming language (with the possible exception of ones designed to be compiled to silicon) allows only a statically fixed set of mutable locations. When an unbounded number of new references may be allocated dynamically at runtime, a static effect system clearly cannot name them all in

advance. The simple approach of just having one big abstraction for all locations ('the store') and tracking only whether some reading or some writing takes place is still sound, but we would like to be more precise.

In many languages, the existing type system gives a natural way to partition the runtime set of mutable locations into disjoint sets. In an ML-like language, an `int ref` and a `bool ref` are never aliased, so one may obtain a useful increase in precision by indexing read, write and allocation effects by types. Ignoring polymorphism again, we might take

$$\mathcal{E} = \mathbb{P}\{\mathrm{rd}(A), \mathrm{wr}(A), \mathrm{al}(A) \mid A \text{ a type}\}$$

(Note that types and effects are now mutually recursive.)

But we can do even better. Imagine that our language had two quite distinct types of references, say red ones and blue ones, and one always had to say which sort one was creating or accessing. Then clearly a red reference and a blue reference can never alias, we could refine our effect types system to track the colours of references involved in store effects, and we could perform some more transformations (for example commuting an expression which can only write blue integer references with one which only reads red integer references).

In its simplest form, the idea of region inference is to take a typing derivation for a monochrome program and to find a way of colouring each reference type appearing in the derivation subject to preserving the validity of the derivation (so, for example, a function expecting a red reference as an argument can never be applied to a blue one). It should be clear that the aim is to use as many different colours as possible. The colours are conventionally called *regions*, because one can imagine that dynamically all the locations of a given colour are allocated in a particular region of the heap.[21]

So now we have three static concepts: type, effect and region. Each of these can be treated monomorphically, with a subwidget relation or polymorphically. The type and effect discipline described by Talpin and Jouvelot in [TJ94] is polymorphic in all three components and indexes reference effects by both regions and types.

Perhaps the most interesting thing about regions is that we can use them to extend our inference system with a rule in which the effect of the conclusion is *smaller* than the effect of the assumption. Consider the following example

```
fun f x = let val r = ref (x+1)
          in !r
          end
```

[21] Alternatively, one might think that any runtime location will have a unique allocation site in the code and all locations with the same allocation site will share a colour, so one could think of a region as a set of static program points. But this is a less satisfactory view, since more sophisticated systems allow references allocated at the same program point to be in different regions, depending on more dynamic contextual information, such as which functions appear in the call chain.

A simple effect system would assign `f` a type and effect like $\texttt{int} \xrightarrow{\{al,rd\}} \texttt{int}, \emptyset$, which seems reasonable, since it is indeed a functional value which takes integers to integers with a latent effect of allocating and reading. But the fact that `f` has this latent effect is actually completely unobservable, since the only uses of storage it makes are completely private. In this case it is easy to see that `f` is observationally equivalent to the completely pure successor function

```
fun f' x = x+1
```

which means that, provided the use to which we are going to make of effect information respects observational equivalence[22] we could soundly just forget all about the latent effect of `f` and infer the type $\texttt{int} \xrightarrow{\emptyset} \texttt{int}$ for it instead. How do regions help? A simple type, region and effect derivation looks like this

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \texttt{x:int} \vdash \texttt{x+1:int}, \emptyset}}{\Gamma, \texttt{x:int} \vdash \texttt{(ref x+1):int ref}_\rho, \{\mathrm{al}_\rho\}} \qquad \dfrac{\vdots}{\Gamma, \texttt{x:int}, \texttt{r:int ref}_\rho \vdash \texttt{(!r):int}, \{\mathrm{rd}_\rho\}}}{\dfrac{\Gamma, \texttt{x:int} \vdash \texttt{(let r=ref x+1 in !r):int}, \{\mathrm{al}_\rho, \mathrm{rd}_\rho\}}{\Gamma \vdash \texttt{(fn x=>let r=ref x+1 in !r)} : \texttt{int} \xrightarrow{\{\mathrm{al}_\rho, \mathrm{rd}_\rho\}} \texttt{int}, \emptyset}}$$

where ρ is a region. Now this is a valid derivation for *any* choice of ρ; in particular, we can pick ρ to be distinct from any region appearing in Γ. That means that the body of the function does not have any effect involving references imported from its surrounding context. Furthermore, the type of the function body is simply `int`, so whatever the rest of the program does with the result of a call to the function, it cannot have any dependency on the references used to produce it. Such considerations motivate the *effect masking* rule

$$\dfrac{\Gamma \vdash M : A, \varepsilon}{\Gamma \vdash M : A, \varepsilon \setminus \{\mathrm{rd}_\rho, \mathrm{al}_\rho, \mathrm{wr}_\rho \mid \rho \notin \Gamma \wedge \rho \notin A\}}$$

Using this rule before just before typing the abstraction in the derivation above does indeed allow us to type `f` as having no observable latent effect.

One of the most remarkable uses of region analysis is Tofte and Talpin's work on static memory management [TT97]: they assign region-annotated types to every non-base value (rather than just mutable references) in an intermediate language where new lexically-scoped regions are introduced explicitly by a `letregion` ρ `in ...end` construct. For a well-typed and annotated program in this language, no value allocated in region ρ will be referenced again after the end of the `letregion` block introducing ρ. Hence that region of the heap may

[22] This should be the case for justifying optimising transformations or inferring more generous polymorphic types, but might not be in the case of a static analysis tool which helps the programmer reason about, say, memory usage.

be safely reclaimed on exiting the block. This technique has been successfully applied in a version of the ML Kit compiler in which there is no runtime garbage collector at all. For some programs, this scheme leads to dramatic reductions in runtime space usage compared with traditional garbage collection, whereas for others the results are much worse. Combining the two techniques is possible, but requires some care, since the region-based memory management reclaims memory which will not be referenced again, but to which there may still be pointers accessible from the GC root. The GC therefore needs to avoid following these 'dangling pointers'.

The soundness of effect masking in the presence of higher-type references and of region-based memory management is not at all trivial to prove. Both [TJ94] and [TT97] formulate correctness in terms of a coinductively defined consistency relation between stores and typing judgements. A number of researchers have recently published more elementary proofs of the correctness of region calculi, either by translation into other systems [BHR99,dZG00] or by more direct methods [HT00,Cal01].

10 Monads and Effect Systems

10.1 Introduction

This section describes how type and effect analyses can be presented in terms of monads and the computational metalanguage. Although this is actually rather obvious, it was only recently that anybody got around to writing anything serious about it. In ICFP 1998, Wadler published a paper [Wad98] (later extended and corrected as [WT99]) showing the equivalence of a mild variant of the effect system of Talpin and Jouvelot [TJ94] and a version of the computational metalanguage in which the computation type constructor is indexed by effects. In the same conference, Benton, Kennedy and Russell described the MLj compiler [BKR98] and its intermediate language MIL, which is a similar effect-refined version of the computational metalanguage. Also in 1998, Tolmach proposed an intermediate representation with a hierarchy of monadic types for use in compiling ML by transformation [Tol98].

The basic observation is that the places where the computation type constructor appears in the call-by-value translation of the lambda calculus into $\lambda\mathrm{ML}_T$ correspond precisely to the places where effect annotations appear in type and effect systems. Effect systems put an ε over each function arrow and on the right-hand side of turnstiles, whilst the CBV translation adds a T to the *end* of each function arrow and on the right hand side of turnstiles. Wadler started with a CBV lambda calculus with a value-polymorphic type, region and effect system tracking store effects (without effect masking). He then showed that Moggi's CBV translation of this language into a version of the metalanguage in which the computation type constructor is annotated with a set of effects (and the monadic `let` rule unions these sets) preserves typing, in that

$$\Gamma \vdash_{eff} M : A, \varepsilon \Rightarrow \Gamma^v \vdash_{mon} M^v : T_\varepsilon(A^v)$$

where

$$\begin{aligned} \texttt{int}^v &= \texttt{int} \\ (A \xrightarrow{\varepsilon} B)^v &= A^v \to T_\varepsilon(B^v) \end{aligned}$$

Wadler also defined an instrumented operational semantics for each of the two languages and used these to prove subject reduction type soundness results in the style of Wright and Felleisen [WF94]. The instrumented operational semantics records not only the evaluation of an expression and a state to a value and a new state, but also a *trace* of the side effects which occur during the evaluation; part of the definition of type soundness is then that when an expression has a static effect ε, any effect occuring in the dynamic trace of its evaluation must be contained in ε.

Tolmach's intermediate language has four monads:

1. The identity monad, used for pure, terminating computations;
2. The lifting monad, used to model potential non-termination;
3. The monad of exceptions and non-termination;
4. The ST monad, which combines lifting, exceptions and the possibility of performing output.

These are linearly ordered, with explicit monad morphisms used to coerce computations from one monad type to a larger one. Tolmach gives a denotational semantics for his intermediate language (using cpos) and presents a number of useful transformation laws which can be validated using this semantics.

10.2 MIL-lite: Monads in MLj

MIL-lite is a simplified fragment of MIL, the intermediate language used in the MLj compiler. It was introduced by Benton and Kennedy in [BK99] as a basis for proving the soundness of some of the effect-based optimizing transformations performed by MLj. Compared with many effect systems in the literature, MIL only performs a fairly crude effect analysis – it doesn't have regions, effect polymorphism or masking. MIL-lite further simplifies the full language by omitting type polymorphism, higher-type references and recursive types as well as various lower level features. Nevertheless, MIL-lite is far from trivial, combining higher-order functions, recursion, exceptions and dynamically allocated state with effect-indexed computation types and subtyping.

Types and terms MIL-lite is a compiler intermediate language for which we first give an operational semantics and then *derive* an equational theory, so there are a couple of design differences between it and Moggi's equational metalanguage. The first is that types are *stratified* into value types (ranged over by τ) and computation types (ranged over by γ); we will have no need of computations of computations. The second difference is that the distinction between computations and values is alarmingly syntactic: the only expressions of value types are normal forms. It is perhaps more elegant to assign value types to a wider

collection of pure expressions than just those in normal form. That is the way Wadler's effect-annotated monadic language is presented, and it leads naturally to a stratified operational semantics in which there is one relation defining the pure reduction of expressions of value type to normal form and another defining the possibly side-effecting evaluation of computations.

Given a countable set $\mathbb{E}$ of exception names, MIL-lite types are defined by

$$\begin{array}{ll} \tau ::= \mathrm{unit} \mid \mathrm{int} \mid \mathrm{intref} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \gamma & \\ \gamma ::= \mathbf{T}_\varepsilon(\tau) & \varepsilon \subseteq \mathcal{E} = \{\bot, r, w, a\} \uplus \mathbb{E} \end{array}$$

We write bool for unit + unit. Function types are restricted to be from values to computations as this is all we shall need to interpret a CBV source language. The effects which we detect are possible failure to terminate ($\bot$), $r$eading from a reference, $w$riting to a reference, $a$llocating a new reference cell and raising a particular exception $E \in \mathbb{E}$. Inclusion on sets of effects induces a subtyping relation:

$$\frac{}{\tau \leqslant \tau} \quad \tau \in \{\mathrm{unit}, \mathrm{int}, \mathrm{intref}\} \qquad \frac{\varepsilon \subseteq \varepsilon' \qquad \tau \leqslant \tau'}{\mathbf{T}_\varepsilon(\tau) \leqslant \mathbf{T}_{\varepsilon'}(\tau')}$$

$$\frac{\tau_1 \leqslant \tau_1' \qquad \tau_2 \leqslant \tau_2'}{\tau_1 \times \tau_2 \leqslant \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \leqslant \tau_1' \qquad \tau_2 \leqslant \tau_2'}{\tau_1 + \tau_2 \leqslant \tau_1' + \tau_2'} \qquad \frac{\tau' \leqslant \tau \qquad \gamma \leqslant \gamma'}{\tau \to \gamma \leqslant \tau' \to \gamma'}$$

Reflexivity and transitivity are consequences of these rules.

There are two forms of typing judgment: $\Gamma \vdash V : \tau$ for values and $\Gamma \vdash M : \gamma$ for computations, where in both cases Γ is a finite map from term variables to *value* types (because the source language is CBV). We assume a countable set $\mathbb{L}$ of locations. The typing rules are shown in Figure 5 and satisfy the usual weakening, strengthening and substitution lemmas. We will sometimes use G to range over both value and computation terms and σ to range over both value and computation types. Most of the terms are unsurprising, but we do use a novel construct

$$\mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ \{E_1.M_1, \ldots, E_n.M_n\}\ \mathrm{in}\ N$$

which should be read "Evaluate the expression M. If successful, bind the result to x and evaluate N. Otherwise, if exception E_i is raised, evaluate the exception handler M_i instead, or if no handler is applicable, pass the exception on." A full discussion of the reasons for adopting the try-handle construct may be found in [BK01], but for now observe that it nicely generalises both handle and Moggi's monadic let, as illustrated by some of the syntactic sugar defined in Figure 6.

For ease of presentation the handlers are treated as a set in which no exception E appears more than once. We let H range over such sets, and write $H \setminus E$ to denote H with the handler for E removed (if it exists). We sometimes use map-like notation, for example writing $H(E)$ for the term M in a handler $E.M \in H$, and writing $\mathrm{dom}(H)$ for $\{E \mid E.M \in H\}$. We write $\Gamma \vdash H : \gamma$ to mean that for all $E.M \in H$, $\Gamma \vdash M : \gamma$.

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \quad \frac{}{\Gamma \vdash \underline{n}:\mathrm{int}} \quad \frac{}{\Gamma \vdash ():\mathrm{unit}} \quad \frac{}{\Gamma \vdash \underline{\ell}:\mathrm{intref}} \quad \ell \in \mathbb{L}$$

$$\frac{\Gamma \vdash V:\tau_i}{\Gamma \vdash \mathrm{in}_i V:\tau_1+\tau_2} \quad i=1,2 \qquad \frac{\Gamma \vdash V_1:\tau_1 \quad \Gamma \vdash V_2:\tau_2}{\Gamma \vdash (V_1,V_2):\tau_1\times\tau_2}$$

$$\frac{\Gamma, x:\tau, f:\tau\to\mathbf{T}_{\varepsilon\cup\{\bot\}}(\tau') \vdash M:\mathbf{T}_\varepsilon(\tau')}{\Gamma \vdash (\mathrm{rec}\, f\ x = M):\tau\to\mathbf{T}_\varepsilon(\tau')} \qquad \frac{\Gamma \vdash V:\tau_1}{\Gamma \vdash V:\tau_2} \quad \tau_1 \leqslant \tau_2$$

$$\frac{\Gamma \vdash V_1:\tau\to\gamma \quad \Gamma \vdash V_2:\tau}{\Gamma \vdash V_1\ V_2:\gamma} \qquad \frac{\Gamma \vdash V:\tau}{\Gamma \vdash \mathrm{val}\, V:\mathbf{T}_\emptyset(\tau)}$$

$$\frac{\Gamma \vdash M:\mathbf{T}_\varepsilon(\tau) \quad \Gamma \vdash H:\mathbf{T}_{\varepsilon'}(\tau') \quad \Gamma, x:\tau \vdash N:\mathbf{T}_{\varepsilon'}(\tau')}{\Gamma \vdash \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ H\ \mathrm{in}\ N:\mathbf{T}_{\varepsilon\setminus\mathrm{dom}(H)\cup\varepsilon'}(\tau')} \qquad \frac{}{\Gamma \vdash \mathrm{raise}\, E:\mathbf{T}_{\{E\}}(\tau)}$$

$$\frac{\Gamma \vdash V:\tau_1\times\tau_2}{\Gamma \vdash \pi_i V:\mathbf{T}_\emptyset(\tau_i)} \quad i=1,2 \qquad \frac{\Gamma \vdash V:\tau_1+\tau_2 \quad \{\Gamma, x_i:\tau_i \vdash M_i:\gamma\}_{i=1,2}}{\Gamma \vdash (\mathrm{case}\, V\ \mathrm{of}\ \mathrm{in}_1 x_1.M_1\ ;\ \mathrm{in}_2 x_2.M_2):\gamma}$$

$$\frac{\Gamma \vdash V:\mathrm{int}}{\Gamma \vdash \mathrm{ref}\, V:\mathbf{T}_{\{a\}}(\mathrm{intref})} \qquad \frac{\Gamma \vdash V:\mathrm{intref}}{\Gamma \vdash !V:\mathbf{T}_{\{r\}}(\mathrm{int})} \qquad \frac{\Gamma \vdash V_1:\mathrm{intref} \quad \Gamma \vdash V_2:\mathrm{int}}{\Gamma \vdash V_1 := V_2:\mathbf{T}_{\{w\}}(\mathrm{unit})}$$

$$\frac{\Gamma \vdash V_1:\mathrm{int} \quad \Gamma \vdash V_2:\mathrm{int}}{\Gamma \vdash V_1+V_2:\mathbf{T}_\emptyset(\mathrm{int})} \qquad \frac{\Gamma \vdash V_1:\mathrm{int} \quad \Gamma \vdash V_2:\mathrm{int}}{\Gamma \vdash V_1=V_2:\mathbf{T}_\emptyset(\mathrm{bool})} \qquad \frac{\Gamma \vdash M:\gamma_1}{\Gamma \vdash M:\gamma_2} \quad \gamma_1 \leqslant \gamma_2$$

Fig. 5. Typing rules for MIL-lite

$\lambda x.\, M$	$\stackrel{\mathrm{def}}{=} \mathrm{rec}\, f\ x = M \quad (f \notin FV(M))$
Ω	$\stackrel{\mathrm{def}}{=} (\mathrm{rec}\, f\ x = f\ x)\ ()$
false	$\stackrel{\mathrm{def}}{=} \mathrm{in}_1()$
true	$\stackrel{\mathrm{def}}{=} \mathrm{in}_2()$
if V then M_2 else M_1	$\stackrel{\mathrm{def}}{=} \mathrm{case}\, V\ \mathrm{of}\ \mathrm{in}_1 x_1.M_1; \mathrm{in}_2 x_2.M_2 \quad (x_i \notin FV(M_i))$
let $x \Leftarrow M$ in N	$\stackrel{\mathrm{def}}{=} \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ \{\}\ \mathrm{in}\ N$
let $x_1 \Leftarrow M_1; x_2 \Leftarrow M_2$ in N	$\stackrel{\mathrm{def}}{=} \mathrm{let}\ x_1 \Leftarrow M_1\ \mathrm{in\ let}\ x_2 \Leftarrow M_2\ \mathrm{in}\ N$
$M; N$	$\stackrel{\mathrm{def}}{=} \mathrm{let}\ x \Leftarrow M\ \mathrm{in}\ N \quad (x \notin FV(N))$
M handle H	$\stackrel{\mathrm{def}}{=} \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ H\ \mathrm{in\ val}\ x$
set $\{\ell_1 \mapsto n_1, \ldots, \ell_k \mapsto n_k\}$	$\stackrel{\mathrm{def}}{=} \underline{\ell_1} := \underline{n_1}\ ;\ \ldots\ ;\ \underline{\ell_k} := \underline{n_k}\ ;\ \mathrm{val}\ ()$
assert (ℓ, n)	$\stackrel{\mathrm{def}}{=} \mathrm{let}\ v \Leftarrow !\underline{\ell}; b \Leftarrow (v = \underline{n})\ \mathrm{in\ if}\ b\ \mathrm{then\ val}\ ()\ \mathrm{else}\ \Omega$
assert $\{\ell_1 \mapsto n_1, \ldots, \ell_k \mapsto n_k\}$	$\stackrel{\mathrm{def}}{=} \mathrm{assert}\ (\ell_1, n_1)\ ;\ \ldots\ ;\ \mathrm{assert}\ (\ell_k, n_k)\ ;\ \mathrm{val}\ ()$

Fig. 6. Syntactic sugar

The analysis The way in which the MIL-lite typing rules express a simple effects analysis should be fairly clear, though some features may deserve further comment. The $\rightarrow$ introduction rule incorporates an extremely feeble, but nonetheless very useful, termination test: the more obvious rule would insist that $\bot \in \varepsilon$, but that would prevent $\lambda x.M$ from getting the natural derived typing rule and would cause undesirable non-termination effects to appear in, particularly, curried recursive functions.

Just as with traditional effect systems, the use of subtyping increases the accuracy of the analysis compared with one which just uses simple types or subeffecting.

There are many possible variants of the rules. For example, there is a stronger (try) rule in which the effects of the handlers are not all required to be the same, and only the effects of handlers corresponding to exceptions occurring in ε are unioned into the effect of the whole expression.

Exercise 59. Give examples which validate the claim that the $\rightarrow$ introduction rule gives better results than the obvious version with $\bot \in \varepsilon$.

MIL-lite does not include recursive types or higher-type references, because they would make proving correctness significantly more difficult. But can you devise candidate rules for an extended language which does include these features? They're not entirely obvious (especially if one tries to make the rules reasonably precise too). It may help to consider

```
datatype U = L of U->U
```

and

```
let val r = ref (fn () => ())
    val _ = r := (fn () => !r ())
in !r
end
```

Operational semantics We present the operational semantics of MIL-lite using a big-step evaluation relation $\Sigma, M \Downarrow \Sigma', R$ where R ranges over value terms and exception identifiers and $\Sigma \in \text{States} \stackrel{\text{def}}{=} \mathbb{L} \rightharpoonup_{\text{fin}} \mathbb{Z}$.

Write $\Sigma, M \Downarrow$ if $\Sigma, M \Downarrow \Sigma', R$ for some Σ', R and $\lfloor G \rfloor$ for the set of location names occuring in G. If $\Sigma, \Delta \in \text{States}$ then $(\Sigma \triangleleft \Delta) \in \text{States}$ is defined by $(\Sigma \triangleleft \Delta)(\ell) = \Delta(\ell)$ if that's defined and $\Sigma(\ell)$ otherwise.

In [BK99], we next prove a number of technical results about the operational semantics, using essentially the techniques described by Pitts in his lectures [Pit00a]. Since most of that material is not directly related to monads or effects, we will omit it from this account, but the important points are the following:

- We are interested in reasoning about *contextual equivalence*, which is a *type-indexed* relation between terms *in context*:

$$\Gamma \vdash G =_{\text{ctx}} G' : \sigma$$

$$\Sigma, \mathrm{val}\, V \Downarrow \Sigma, V \qquad \Sigma, \mathrm{raise}\, E \Downarrow \Sigma, E \qquad \Sigma, \pi_i(V_1, V_2) \Downarrow \Sigma, V_i$$

$$\Sigma, \underline{n} + \underline{m} \Downarrow \Sigma, \underline{n+m} \qquad \Sigma, \underline{n} = \underline{n} \Downarrow \Sigma, \mathrm{true} \qquad \Sigma, \underline{n} = \underline{m} \Downarrow \Sigma, \mathrm{false}\ (n \neq m)$$

$$\Sigma, !\underline{\ell} \Downarrow \Sigma, \underline{\Sigma(\ell)} \qquad \Sigma, \underline{\ell} := \underline{n} \Downarrow \Sigma[\ell \mapsto n], () \qquad \Sigma, \mathrm{ref}\, \underline{n} \Downarrow \Sigma \uplus [\ell \mapsto n], \underline{\ell}$$

$$\frac{\Sigma, M_i[V/x_i] \Downarrow \Sigma', R}{\Sigma, \mathrm{case}\ \mathrm{in}_i V\ \mathrm{of}\ \ \mathrm{in}_1 x_1.M_1\ ;\ \mathrm{in}_2 x_2.M_2 \Downarrow \Sigma', R} \quad i = 1, 2$$

$$\frac{\Sigma, M[V/x, (\mathrm{rec}\, f\ x = M)/f] \Downarrow \Sigma', R}{\Sigma, (\mathrm{rec}\, f\ x = M)\ V \Downarrow \Sigma', R} \qquad \frac{\Sigma, M \Downarrow \Sigma', V \quad \Sigma', N[V/x] \Downarrow \Sigma'', R}{\Sigma, \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ H\ \mathrm{in}\ N \Downarrow \Sigma'', R}$$

$$\frac{\Sigma, M \Downarrow \Sigma', E \quad \Sigma', M' \Downarrow \Sigma'', R}{\Sigma, \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ H\ \mathrm{in}\ N \Downarrow \Sigma'', R} \quad H(E) = M'$$

$$\frac{\Sigma, M \Downarrow \Sigma', E}{\Sigma, \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\ H\ \mathrm{in}\ N \Downarrow \Sigma', E} \quad E \notin \mathrm{dom}(H)$$

Fig. 7. Evaluation relation for MIL-lite

- Rather than work with contextual equivalence directly, we show that contextual equivalence coincides with *ciu equivalence*, which shows that only certain special contexts need be considered to establish equivalence. For MIL-lite, ciu equivalence is the open extension of the relation defined by the following clauses:
  - If $M_1 : \mathbf{T}_\varepsilon(\tau)$ and $M_2 : \mathbf{T}_\varepsilon(\tau)$ we write $M_1 \approx M_2 : \mathbf{T}_\varepsilon(\tau)$ and say M_1 *is ciu equivalent to* M_2 *at type* $\mathbf{T}_\varepsilon(\tau)$ when $\forall N, H$ such that $x : \tau \vdash N : \gamma$ and $\vdash H : \gamma$, and $\forall \Sigma \in \mathrm{States}$ such that $\mathrm{dom}\, \Sigma \supseteq \lfloor M_1, M_2, H, N \rfloor$ we have

    $$\Sigma, \mathrm{try}\ x \Leftarrow M_1\ \mathrm{catch}\ H\ \mathrm{in}\ N \Downarrow \quad \Leftrightarrow \quad \Sigma, \mathrm{try}\ x \Leftarrow M_2\ \mathrm{catch}\ H\ \mathrm{in}\ N \Downarrow$$

  - If $V_1 : \tau$ and $V_2 : \tau$ then we write $V_1 \approx V_2 : \tau$ for $\mathrm{val}\, V_1 \approx \mathrm{val}\, V_2 : \mathbf{T}_\emptyset(\tau)$.

10.3 Transforming MIL-lite

Semantics of Effects We want to use the effect information expressed in MIL-lite types to justify some optimizing transformations. Our initial inclination was to prove the correctness of these transformations by using a denotational semantics. However, giving a good denotational semantics of MIL-lite is surprisingly tricky, not really because of the multiple computational types, but because of the presence of dynamically allocated references. Stark's thesis [Sta94] examines equivalence in a very minimal language with dynamically generated names in considerable detail and does give a functor category semantics for a language

with higher order functions and integer references. But MIL-lite is rather more complex than Stark's language, requiring a functor category into cpos (rather than sets) and then indexed monads over that. Worst of all, the resulting semantics turns out to be very far from fully abstract – it actually fails to validate some of the most elementary transformations which we wished to perform. So we decided to prove correctness of our transformations using operational techniques instead.

Most work on using operational semantics to prove soundness of effect analyses involves instrumenting the semantics to trace computational effects in some way and then proving that 'well-typed programs don't go wrong' in this modified semantics. This approach is perfectly correct, but the notion of correctness and the meaning of effect annotations is quite intensional and closely tied to the formal system used to infer them. Since we wanted to prove the soundness of using the analysis to justify observational equivalences in an uninstrumented semantics, we instead tried to characterise the meaning of effect-annotated types as properties of terms which are closed under observational equivalence in the uninstrumented semantics. To give a simple example of the difference between the two approaches, a weak effect system (such as that in MIL-lite) will only assign a term an effect which does not contain w if the evaluation of that term really does never perform a write operation. A region-based analysis may infer such an effect if it can detect that the term only writes to private locations. But the property we *really* want to use to justify equations is much more extensional: it's that after evaluating the term, the contents of all the locations which were allocated before the evaluation are indistinguishable from what they were to start with.

The decision not to use an instrumented semantics is largely one of taste, but there is another (post hoc) justification. There are a few places in the MLj libraries where we manually annotate bindings with smaller effect types than could be inferred by our analysis, typically so that the rewrites can dead-code them if they are not used (for example, the initialisation of lookup tables used in the floating point libraries). Since those bindings *do* have the extensional properties associated with the type we force them to have, the correctness result for our optimizations extends easily to these manually annotated expressions.

We capture the intended meaning $[\![\sigma]\!]$ of each type σ in MIL-lite as the set of closed terms of that type which pass all of a collection of cotermination tests $\mathrm{Tests}_\sigma \subseteq \mathrm{States} \times \mathrm{Ctxt}_\sigma \times \mathrm{Ctxt}_\sigma$ where Ctxt_σ is the set of closed contexts with a finite number of holes of type σ. Formally:

$$[\![\sigma]\!] \stackrel{\mathrm{def}}{=} \{\, G : \sigma \mid \forall (\Sigma, M[\cdot], M'[\cdot]) \in \mathrm{Tests}_\sigma . \\ \lfloor M[G], M'[G] \rfloor \subseteq \mathrm{dom}\, \Sigma \Rightarrow (\Sigma, M[G] \Downarrow \leftrightarrow \Sigma, M'[G] \Downarrow) \,\}$$

We define Tests_σ inductively as shown in Figure 8.

$$\mathrm{Tests}_{\mathrm{int}} \stackrel{\mathrm{def}}{=} \{\} \qquad \mathrm{Tests}_{\mathrm{intref}} \stackrel{\mathrm{def}}{=} \{\} \qquad \mathrm{Tests}_{\mathrm{unit}} \stackrel{\mathrm{def}}{=} \{\}$$

$$\mathrm{Tests}_{\tau_1 \times \tau_2} \stackrel{\mathrm{def}}{=} \bigcup_{i=1,2} \{(\Sigma, M[\pi_i[\cdot]], M'[\pi_i[\cdot]]) \mid (\Sigma, M[\cdot], M'[\cdot]) \in \mathrm{Tests}_{\tau_i}\}$$

$$\mathrm{Tests}_{\tau_1 + \tau_2} \stackrel{\mathrm{def}}{=} \bigcup_{i=1,2} \{(\Sigma, \mathrm{case}\ [\cdot]\ \mathrm{of}\ \mathrm{in}_i\, x.M[x]\ ;\ \mathrm{in}_{3-i}\, y.\Omega,\ \mathrm{case}\ [\cdot]\ \mathrm{of}\ \mathrm{in}_i\, x.M'[x]\ ;\ \mathrm{in}_{3-i}\, y.\Omega) \mid (\Sigma, M[\cdot], M'[\cdot]) \in \mathrm{Tests}_{\tau_i}\}$$

$$\mathrm{Tests}_{\tau \to \gamma} \stackrel{\mathrm{def}}{=} \{(\Sigma, M[[\cdot]\ V], M'[[\cdot]\ V]) \mid V \in [\![\tau]\!],\ (\Sigma, M[\cdot], M'[\cdot]) \in \mathrm{Tests}_{\gamma}\}$$

$$\mathrm{Tests}_{\mathbf{T}_\varepsilon \tau} \stackrel{\mathrm{def}}{=} \{(\Sigma, \mathrm{let}\ x \Leftarrow [\cdot]\ \mathrm{in\ set}\ \Sigma'; M[x], \mathrm{let}\ x \Leftarrow [\cdot]\ \mathrm{in\ set}\ \Sigma'; M'[x]) \mid (\Sigma', M[\cdot], M'[\cdot]) \in \mathrm{Tests}_{\tau},\ \Sigma \in \mathrm{States}\} \cup \bigcup_{e \notin \varepsilon} \mathrm{Tests}_{\overline{e},\tau}$$

where

$$\mathrm{Tests}_{\overline{\perp},\tau} \stackrel{\mathrm{def}}{=} \{(\Sigma, [\cdot], \mathrm{val}\,()) \mid \Sigma \in \mathrm{States}\}$$

$$\mathrm{Tests}_{\overline{w},\tau} \stackrel{\mathrm{def}}{=} \{(\Sigma, \mathrm{let}\ y \Leftarrow !\underline{\ell}\ \mathrm{in\ try}\ x \Leftarrow [\cdot]\ \mathrm{catch}\ E.M\ \mathrm{in}\ N,\ \mathrm{try}\ x \Leftarrow [\cdot]\ \mathrm{catch}\ E.\mathrm{let}\ y \Leftarrow !\underline{\ell}\ \mathrm{in}\ M\ \mathrm{in\ let}\ y \Leftarrow !\underline{\ell}\ \mathrm{in}\ N) \mid y : \mathrm{int}, x : \tau \vdash N : \gamma, y : \mathrm{int} \vdash M : \gamma, \Sigma \in \mathrm{States}, \ell \in \mathrm{dom}\,\Sigma\}$$

$$\mathrm{Tests}_{\overline{r},\tau} \stackrel{\mathrm{def}}{=} \{(\Sigma, \mathrm{d}(\Sigma, \Delta, E); \mathrm{try}\ x \Leftarrow [\cdot]\ \mathrm{catch}\ E.\mathrm{assert}\ \Sigma \triangleleft \Delta; \mathrm{raise}\ E\ \mathrm{in}\ N,\ \mathrm{d}(\Sigma, \Delta, E); \underline{\ell} := \underline{n}; \mathrm{try}\ x \Leftarrow [\cdot]\ \mathrm{catch}\ E.\mathrm{assert}\ \Sigma[\ell \mapsto n] \triangleleft \Delta; \mathrm{raise}\ E\ \mathrm{in\ assert}\ (\ell, (\Sigma[\ell \mapsto n] \triangleleft \Delta)(\ell)); \underline{\ell} := \overline{(\Sigma \triangleleft \Delta)(\ell)}; N) \mid E \in \mathbb{E}, \Sigma, \Delta \in \mathrm{States}, \mathrm{dom}\,\Delta \subseteq \mathrm{dom}\,\Sigma \ni \ell, n \in \mathbb{Z}, x : \tau \vdash N : \gamma\} \cup \{(\Sigma, [\cdot]\ \mathrm{handle}\ E.\Omega, \mathrm{set}\ \Sigma'; [\cdot]\ \mathrm{handle}\ E.\Omega) \mid \Sigma, \Sigma' \in \mathrm{States}, E \in \mathbb{E}\}$$

$$\mathrm{Tests}_{\overline{E},\tau} \stackrel{\mathrm{def}}{=} \{(\Sigma, [\cdot], [\cdot]\ \mathrm{handle}\ E.N) \mid \Sigma \in \mathrm{States}, \vdash N : \gamma\}$$

$$\mathrm{Tests}_{\overline{a},\tau} \stackrel{\mathrm{def}}{=} \{(\Sigma, \mathrm{let}\ x \Leftarrow [\cdot]; y \Leftarrow (\mathrm{set}\ \Sigma; [\cdot])\ \mathrm{in}\ N, \mathrm{let}\ x \Leftarrow [\cdot]; y \Leftarrow \mathrm{val}\, x\ \mathrm{in}\ N) \mid \Sigma \in \mathrm{States}, x : \tau, y : \tau \vdash N : \gamma\}$$

and

$$\mathrm{K}_\Sigma n \stackrel{\mathrm{def}}{=} \{\ell \mapsto n \mid \ell \in \mathrm{dom}(\Sigma)\}$$

$$\mathrm{d}(\Sigma, \Delta, E) \stackrel{\mathrm{def}}{=} \mathrm{set}\ \mathrm{K}_\Sigma 0; (([\cdot]; \mathrm{val}\,())\ \mathrm{handle}\ E.\mathrm{val}\,()); \mathrm{assert}\ \mathrm{K}_\Sigma 0 \triangleleft \Delta; \mathrm{set}\ \mathrm{K}_\Sigma 1; (([\cdot]; \mathrm{val}\,())\ \mathrm{handle}\ E.\mathrm{val}\,()); \mathrm{assert}\ \mathrm{K}_\Sigma 1 \triangleleft \Delta; \mathrm{set}\ \Sigma$$

Fig. 8. Definition of Tests_σ

Although these definitions appear rather complex, at value types they actually amount to a familiar-looking logical predicate:

Lemma 101

- $[\![\mathit{int}]\!] = \{\underline{n} \mid n \in \mathbb{Z}\}$, $[\![\mathit{intref}]\!] = \{\underline{\ell} \mid \ell \in \mathbb{L}\}$ *and* $[\![\mathit{unit}]\!] = \{()\}$.
- $[\![\tau_1 \times \tau_2]\!] = \{(V_1, V_2) \mid V_1 \in [\![\tau_1]\!], V_2 \in [\![\tau_2]\!]\}$
- $[\![\tau \to \gamma]\!] = \{F : \tau \to \gamma \mid \forall V \in [\![\tau]\!].(F\ V) \in [\![\gamma]\!]\}$
- $[\![\tau_1 + \tau_2]\!] = \bigcup_{i=1,2}\{in_i V \mid V \in [\![\tau_i]\!]\}$

□

Lemma 102 *If* $\sigma \leqslant \sigma'$ *then* $[\![\sigma]\!] \subseteq [\![\sigma']\!]$. □

We also have to prove an operational version of admissibility for the predicate associated with each type. This follows from a standard 'compactness of evaluation' or 'unwinding' result which is proved using termination induction, but we omit the details. Finally, we can prove the 'Fundamental Theorem' for our logical predicate, which says that the analysis is correct in the sense that whenever a term is given a particular type it actually satisfies the property associated with that type:

Theorem 60. *If* $\boldsymbol{x_i} : \boldsymbol{\tau_i} \vdash G : \sigma$ *and* $\boldsymbol{V_i} \in [\![\boldsymbol{\tau_i}]\!]$ *then* $G[\boldsymbol{V_i}/\boldsymbol{x_i}] \in [\![\sigma]\!]$. □

Although we have explained the meaning of the logical predicate at value types, it seems worth commenting a little further on the definitions of $\mathrm{Tests}_{\overline{e},\tau}$. The intention is that the extent of $\mathrm{Tests}_{\overline{e},\tau}$ is the set of computations of type $\mathbf{T}_{\mathcal{E}}(\tau)$ which definitely do *not* have effect e. So, passing all the tests in $\mathrm{Tests}_{\overline{\perp},\tau}$ is easily seen to be equivalent to not diverging in any state and passing all the tests in $\mathrm{Tests}_{\overline{E},\tau}$ means not throwing exception E in any state.

The tests concerning store effects are a little more subtle. It is not too hard to see that $\mathrm{Tests}_{\overline{w},\tau}$ expresses not *observably* writing the store. Similarly, $\mathrm{Tests}_{\overline{r},\tau}$ tests (contortedly!) for not observably reading the store, by running the computation in different initial states and seeing if the results can be distinguished by a subsequent continuation.

The most surprising definition is probably that of $\mathrm{Tests}_{\overline{a},\tau}$, the extent of which is intended to be those computations which do not observably allocate any new storage locations. This should include, for example, a computation which allocates a reference and then returns a function which uses that reference to keep count of how many times it has been called, but which never reveals the counter, nor returns different results according to its value. However, the definition of $\mathrm{Tests}_{\overline{a},\tau}$ does not seem to say anything about store extension; what it actually captures is those computations for which two evaluations in equivalent initial states yield indistinguishable results. Our choice of this as the meaning of 'doesn't allocate' was guided by the optimising transformations which we wished to be able to perform rather than a deep understanding of exactly what it means to not allocate observably, but in retrospect it seems quite reasonable.

$$\beta\text{-}\times\frac{\Gamma \vdash V_1 : \tau_1 \qquad \Gamma \vdash V_2 : \tau_2}{\Gamma \vdash \pi_i(V_1, V_2) \cong \mathrm{val}\, V_i : \mathbf{T}_\emptyset(\tau_i)} \qquad \beta\text{-}\mathbf{T}\frac{\Gamma \vdash V : \tau \qquad \Gamma, x : \tau \vdash M : \gamma}{\Gamma \vdash \mathrm{let}\ x \Leftarrow \mathrm{val}\, V\ \mathrm{in}\ M \cong M[V/x] : \gamma}$$

$$\beta\text{-}\rightarrow\frac{\Gamma, x : \tau, f : \tau \to \mathbf{T}_{\varepsilon\cup\{\bot\}}(\tau') \vdash M : \mathbf{T}_\varepsilon(\tau') \qquad \Gamma \vdash V : \tau}{\Gamma \vdash (\mathrm{rec}\ f\ x = M)\ V \cong M[V/x, \mathrm{rec}\ f\ x = M/f] : \mathbf{T}_\varepsilon(\tau')}$$

$$\beta\text{-}+\frac{\Gamma \vdash V : \tau_i \qquad \Gamma, x_1 : \tau_1 \vdash M_1 : \gamma \qquad \Gamma, x_2 : \tau_2 \vdash M_2 : \gamma}{\Gamma \vdash \mathrm{case}\ \mathrm{in}_i V\ \mathrm{of}\ \mathrm{in}_1 x_1.M_1; \mathrm{in}_2 x_2.M_2 \cong M_i[V/x_i] : \gamma}$$

$$\eta\text{-}\times\frac{\Gamma \vdash V : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{let}\ x_1 \Leftarrow \pi_1 V; x_2 \Leftarrow \pi_2 V\ \mathrm{in}\ \mathrm{val}\,(x_1, x_2) \cong \mathrm{val}\, V : \mathbf{T}_\emptyset(\tau_1 \times \tau_2)}$$

$$\eta\text{-}+\frac{\Gamma \vdash V : \tau_1 + \tau_2}{\Gamma \vdash \mathrm{case}\ V\ \mathrm{of}\ \mathrm{in}_1 x_1.\mathrm{val}\,(\mathrm{in}_1 x_1); \mathrm{in}_2 x_2.\mathrm{val}\,(\mathrm{in}_2 x_2) \cong \mathrm{val}\, V : \mathbf{T}_\emptyset(\tau_1 + \tau_2)}$$

$$\eta\text{-}\rightarrow\frac{\Gamma \vdash V : \tau \to \gamma}{\Gamma \vdash \mathrm{rec}\ f\ x = V\ x \cong V : \tau \to \gamma} \qquad \eta\text{-}\mathbf{T}\frac{\Gamma \vdash M : \gamma}{\Gamma \vdash \mathrm{let}\ x \Leftarrow M\ \mathrm{in}\ \mathrm{val}\, x \cong M : \gamma}$$

$$cc_1\frac{\Gamma \vdash M_1 : \mathbf{T}_{\varepsilon_1}(\tau_1) \qquad \Gamma, y : \tau_1 \vdash M_2 : \mathbf{T}_{\varepsilon_2}(\tau_2) \qquad \Gamma, y : \tau_1, x : \tau_2 \vdash M_3 : \mathbf{T}_{\varepsilon_3}(\tau_3)}{\Gamma \vdash \mathrm{let}\ x \Leftarrow (\mathrm{let}\ y \Leftarrow M_1\ \mathrm{in}\ M_2)\ \mathrm{in}\ M_3 \cong \mathrm{let}\ y \Leftarrow M_1; x \Leftarrow M_2\ \mathrm{in}\ M_3 : \mathbf{T}_{\varepsilon_1\cup\varepsilon_2\cup\varepsilon_3}(\tau_3)}$$

$$cc_2\frac{\Gamma \vdash V : \tau_1 + \tau_2 \quad \{\Gamma, x_i : \tau_i \vdash M_i : \mathbf{T}_\varepsilon(\tau)\} \quad \Gamma, x : \tau \vdash N : \mathbf{T}_{\varepsilon'}(\tau')}{\Gamma \vdash \mathrm{let}\ x \Leftarrow \mathrm{case}\, V\ \mathrm{of}\ \{\mathrm{in}_i x_i.M_i\}\ \mathrm{in}\ N \cong \mathrm{case}\, V\ \mathrm{of}\ \{\mathrm{in}_i x_i.\mathrm{let}\ x \Leftarrow M_i\ \mathrm{in}\ N\} : \mathbf{T}_{\varepsilon\cup\varepsilon'}(\tau')}$$

$$\beta\text{-}E\frac{\Gamma \vdash M : \gamma \qquad \Gamma \vdash H : \gamma \qquad \Gamma, x : \tau \vdash N : \gamma}{\Gamma \vdash \mathrm{try}\ x \Leftarrow \mathrm{raise}\ E\ \mathrm{catch}\,(E.M); H\ \mathrm{in}\ N \cong M : \gamma}$$

$$\eta\text{-}E\frac{\Gamma \vdash M : \mathbf{T}_\varepsilon(\tau) \qquad \Gamma \vdash H : \mathbf{T}_{\varepsilon'}(\tau') \qquad \Gamma, x : \tau \vdash N : \mathbf{T}_{\varepsilon'}(\tau')}{\Gamma \vdash \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\,(E.\mathrm{raise}\ E); H\ \mathrm{in}\ N \cong \mathrm{try}\ x \Leftarrow M\ \mathrm{catch}\, H\ \mathrm{in}\ N : \mathbf{T}_{\varepsilon\cup\varepsilon'}(\tau')}$$

Fig. 9. Effect-independent equivalences (1)

Effect-independent equivalences Figure 9 presents some typed observational congruences that correspond to identities from the equational theory of the computational lambda calculus, and Figure 10 presents equivalences that involve local side-effecting behaviour.[23] Directed variants of many of these are useful transformations that are in fact performed by MLj (although the duplication of terms in cc_2 is avoided by introducing a special kind of abstraction). These equations can be derived without recourse to our logical predicate, by making use of a rather strong notion of equivalence called *Kleene equivalence* that can easily be shown to be contained in ciu equivalence. Two terms are Kleene equivalent if they coterminate in any initial state with syntactically identical results and the same values in all accessible locations of the store (Mason and Talcott call this 'strong isomorphism' [MT91]).

$$\frac{\Gamma \vdash V : \mathrm{int} \qquad \Gamma \vdash M : \mathbf{T}_{\varepsilon}(\tau)}{\Gamma \vdash \mathrm{let}\ x \Leftarrow \mathrm{ref}\ V\ \mathrm{in}\ M \cong M : \mathbf{T}_{\varepsilon \cup \{a\}}(\tau)}$$

$$\frac{\Gamma \vdash V : \mathrm{intref} \qquad \Gamma, x : \mathrm{int}, y : \mathrm{int} \vdash M : \mathbf{T}_{\varepsilon}(\tau)}{\Gamma \vdash \mathrm{let}\ x \Leftarrow !V; y \Leftarrow !V\ \mathrm{in}\ M \cong \mathrm{let}\ x \Leftarrow !V; y \Leftarrow \mathrm{val}\ x\ \mathrm{in}\ M : \mathbf{T}_{\varepsilon \cup \{r\}}(\tau)}$$

$$\frac{\Gamma \vdash V_1 : \mathrm{int} \qquad \Gamma \vdash V_2 : \mathrm{int} \qquad \Gamma, x_1 : \mathrm{intref}, x_2 : \mathrm{intref} \vdash M : \mathbf{T}_{\varepsilon}(\tau)}{\Gamma \vdash \mathrm{let}\ x_1 \Leftarrow \mathrm{ref}\ V_1; x_2 \Leftarrow \mathrm{ref}\ V_2\ \mathrm{in}\ M \cong \mathrm{let}\ x_2 \Leftarrow \mathrm{ref}\ V_2; x_1 \Leftarrow \mathrm{ref}\ V_1\ \mathrm{in}\ M : \mathbf{T}_{\varepsilon \cup \{a\}}(\tau)}$$

$$\frac{\Gamma \vdash V_1 : \mathrm{intref} \qquad \Gamma \vdash V_2 : \mathrm{int} \qquad \Gamma, x : \mathrm{int} \vdash M : \mathbf{T}_{\varepsilon}(\tau)}{\Gamma \vdash V_1 := V_2; \mathrm{let}\ x \Leftarrow !V_1\ \mathrm{in}\ M \cong V_1 := V_2; M[V_2/x] : \mathbf{T}_{\varepsilon \cup \{r,w\}}(\tau)}$$

Fig. 10. Effect-independent equivalences (2)

The beta-equivalences and commuting conversions of Figure 9 together with the equivalences of Figure 10 are derived directly as Kleene equivalences. Derivation of the eta-equivalences involves first deriving a number of extensionality properties using ciu equivalence; similar techniques are used in [Pit97].

Effect-dependent equivalences We now come to a set of equivalences that are dependent on effect information, which are shown in Figure 11. Notice how the first three of these equations respectively subsume the first three local equivalences of Figure 10. Each of these equivalences is proved by considering evaluation of each side in an arbitrary ciu-context and then using the logical predicate to show that if the evaluation terminates then so does the evaluation of the other side in the same context.

[23] Some side conditions on variables are implicit in our use of contexts. For example, the first equation in Figure 10 has the side condition that $x \notin fv(M)$.

$$\textit{discard}\ \frac{\Gamma \vdash M : \mathbf{T}_{\varepsilon_1}(\tau_1) \qquad \Gamma \vdash N : \mathbf{T}_{\varepsilon_2}(\tau_2)}{\Gamma \vdash \text{let } x \Leftarrow M \text{ in } N \cong N : \mathbf{T}_{\varepsilon_1 \cup \varepsilon_2}\tau_2} \quad \text{where } \varepsilon_1 \subseteq \{r, a\}$$

$$\textit{copy}\ \frac{\Gamma \vdash M : \mathbf{T}_{\varepsilon}(\tau) \qquad \Gamma, x : \tau, y : \tau \vdash N : \mathbf{T}_{\varepsilon'}(\tau')}{\Gamma \vdash \text{let } x \Leftarrow M; y \Leftarrow M \text{ in } N \cong \text{let } x \Leftarrow M; y \Leftarrow \text{val } x \text{ in } N : \mathbf{T}_{\varepsilon \cup \varepsilon'}(\tau')} \quad \text{where } \{r, a\} \cap \varepsilon = \emptyset \text{ or } \{w, a\} \cap \varepsilon = \emptyset$$

$$\textit{swap}\ \frac{\Gamma \vdash M_1 : \mathbf{T}_{\varepsilon_1}(\tau_1) \qquad \Gamma \vdash M_2 : \mathbf{T}_{\varepsilon_2}(\tau_2) \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash N : \mathbf{T}_{\varepsilon_3}(\tau_3)}{\Gamma \vdash \text{let } x_1 \Leftarrow M_1; x_2 \Leftarrow M_2 \text{ in } N \cong \text{let } x_2 \Leftarrow M_2; x_1 \Leftarrow M_1 \text{ in } N : \mathbf{T}_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}(\tau_3)} \quad \text{where } \varepsilon_1, \varepsilon_2 \subseteq \{r, a, \bot\} \text{ or } \varepsilon_1 \subseteq \{a, \bot\}, \varepsilon_2 \subseteq \{r, w, a, \bot\}$$

$$\textit{dead-try}\ \frac{\Gamma \vdash M : \mathbf{T}_{\varepsilon}(\tau) \qquad \Gamma \vdash H : \mathbf{T}_{\varepsilon'}(\tau') \qquad \Gamma, x : \tau \vdash N : \mathbf{T}_{\varepsilon'}(\tau')}{\Gamma \vdash \text{try } x \Leftarrow M \text{ catch } H \text{ in } N \cong \text{try } x \Leftarrow M \text{ catch } H \setminus E \text{ in } N : \mathbf{T}_{\varepsilon \cup \varepsilon'}(\tau')} \quad \text{where } E \notin \varepsilon$$

Fig. 11. Effect-dependent equivalences

10.4 Effect-Dependent Rewriting in MLj

In practice, much of the benefit MLj gets from effect-based rewriting is simply from dead-code elimination (*discard* and *dead-try*). A lot of dead code (particularly straight after linking) is just unused top-level function bindings, and these could clearly be removed by a simple syntactic check instead of a type-based effect analysis. Nevertheless, both unused non-values which detectably at most read or allocate and unreachable exception handlers do occur fairly often too, and it is convenient to be able to use a single framework to eliminate them all. Here is an example (from [BK01]) of how tracking exception effects works together with MIL's unusual handler construct to improve an ML program for summing the elements of an array:

```
fun sumarray a =
  let fun s(n,sofar) = let val v = Array.sub(a,n)
                       in s(n+1, sofar+v)
                       end handle Subscript => sofar
  in s(0,0)
  end
```

Because the SML source language doesn't have *try*, the programmer has made the handler cover both the array access and the recursive call to the inner function `s`. But this would prevent a naïve compiler from recognising that call as tail-recursive. In MLj, the intermediate code for `s` looks like (in MLish, rather than MIL, syntax):

```
fun s(n,sofar) =
      try val x = try val v = Array.sub(a,n)
```

```
                catch {}
                in s(n+1, sofar+v)
                end
        catch Subscript => sofar
        in x
        end
```

A commuting conversion turns this into

```
fun s(n,sofar) = try val v = Array.sub(a,n)
                 catch Subscript => sofar
                 in try val x = s(n+1, sofar+v)
                    catch Subscript => sofar
                    in x
                    end
                 end
```

The effect analysis detects that the recursive call to `s` cannot, in fact, ever throw the `Subscript` exception, so the function is rewritten again to

```
fun s(n,sofar) = try val v = Array.sub(a,n)
                 catch Subscript => sofar
                 in s(n+1, sofar+v)
                 end
```

which *is* tail recursive, and so gets compiled as a loop in the final code for `sumarray`.

Making practical use of the *swap* and *copy* equations is more difficult – although it is easy to come up with real programs which could be usefully improved by sequences of rewrites including those equations, it is hard for the compiler to spot when commuting two computations makes useful progress towards a more significant rewrite. The most significant effect-based code motion transformation which we do perform is pulling constant, pure computations out of functions (in particular, loops), a special case of which is

$$\frac{\Gamma \vdash M : \mathbf{T}_{\emptyset}(\tau_3) \qquad \Gamma, f : \tau_1 \to \mathbf{T}_{\varepsilon \cup \bot}(\tau_2), x : \tau_1, y : \tau_3 \vdash N : \mathbf{T}_{\varepsilon}(\tau_2)}{\Gamma \vdash \mathrm{val}\,(\mathrm{rec}\, f\ x = \mathrm{let}\ y \Leftarrow M\ \mathrm{in}\ N) \cong \mathrm{let}\ y \Leftarrow M\ \mathrm{in}\ \mathrm{val}\,(\mathrm{rec}\, f\ x = N) : \mathbf{T}_{\emptyset}(\tau_1 \to \mathbf{T}_{\varepsilon}(\tau_2))}$$

where there's an implied side condition that neither f nor x is free in M. This is not always an improvement (if the function is never applied), but in the absence of more information it's worth doing anyway. Slightly embarassingly, this is not an equivalence which we have proved correct using the techniques described here, however.

One other place where information about which expressions commute could usefully be applied is in a compiler backend, for example in register allocation. We haven't tried this in MLj since a JIT compiler will do its *own* job of allocating real machine registers and scheduling real machine instructions later, which makes doing a very 'good' job of compiling virtual machine code unlikely to produce great improvements in the performance of the final machine code.

An early version of the compiler also implemented a type-directed uncurrying transformation, exploiting the isomorphism

$$\tau_1 \to \mathbf{T}_\emptyset(\tau_2 \to \mathbf{T}_\varepsilon(\tau_3)) \;\cong\; \tau_1 \times \tau_2 \to \mathbf{T}_\varepsilon(\tau_3)$$

but this can lead to extra work being done if the function is actually partially applied, so this transformation also seems to call for auxiliary information to be gathered.

10.5 Effect Masking and Monadic Encapsulation

We have seen that it is not too hard to recast simple effect systems in a monadic framework. But what is the monadic equivalent of effect masking? The answer is something like the encapsulation of side-effects provided by `runST` in Haskell, but the full connection has not yet been established.

Haskell allows monadic computations which make purely local use of state to be encapsulated as values with 'pure' types by making use of a cunning trick with type variables which is very similar to the use of regions in effect systems. Briefly (see Section 5 for more information), the state monad is parameterized not only by the type of the state s, but also by another 'dummy' type variable r.[24] In MLish syntax:

```
datatype ('r,'s,'a) ST = S of 's -> 's * 'a
```

The idea is that the r parameters of types inferred for computations whose states might interfere will be unified, so if a computation can be assigned a type which is parametrically polymorphic in r, then its use of state can be encapsulated. This is expressed using the `runST` combinator which is given the rank-2 polymorphic type

$$\texttt{runST} : \forall s, a.(\forall r.(r, s, a)\texttt{ST}) \to a$$

Just as the soundness of effect masking and of the region calculus is hard to establish, proving the correctness of monadic encapsulation is difficult. Early attempts to prove soundness by subject reduction [LS97] are now known to be incorrect.

More recently, Semmelroth and Sabry have succeeded in defining a CBV language with monadic encapsulation, relating this to a language with effect masking and proving type soundness [SS99]. Moggi and Palumbo have also addressed this problem [MP99], by defining a slightly different form of monadic encapsulation (without the 'bogus' type parameter) and proving a type soundness result for a language in which the stateful operations are strict. Proving soundness for monadic encapsulation in a CBN language with lazy state operations is still, so far as I am aware, an open problem.

[24] Actually, Haskell's built-in state monad is not parameterized on the type of the state itself.

11 Curry-Howard Correspondence and Monads

This section provides a little optional background on a logical reading of the computational metalanguage and explains the term 'commuting conversion'.

Most readers will have some familiarity with the so-called Curry-Howard Correspondence (or Isomorphism, aka the Propositions-as-Types Analogy). This relates types in certain typed lambda calculi to propositions in intuitionistic logics, typed terms in context to (natural deduction) proofs of propositions from assumptions, and reduction to proof normalization. The basic example of the correspondence relates the simply typed lambda calculus with function, pair and disjoint union types to intutionisitic propositional logic with implication, conjunction and disjunction [GLT89].

Whilst it may be true that almost no realistic programming language corresponds accurately to anything which might plausibly be called a logic (because of the presence of general recursion, if nothing else), logic and proof theory can still provide helpful insights into the design of programming languages and intermediate languages. Partly this seems to be because proof theorists have developed a number of taxonomies and criteria for 'well-behavedness' of proof rules which turn out to be transferable to the design of 'good' language syntax.

The computational metalanguage provides a nice example of the applicability of proof theoretic ideas (see [BBdP98] for details). If one reads the type rules for the introduction and elimination of the computation type constructor logically, then one ends up with an intuitionistic *modal* logic (which we dubbed 'CL-logic') with a slightly unusual kind of possibility modality, $\diamond$. In sequent-style natural deduction form:

$$\frac{\Gamma \vdash A}{\Gamma \vdash \diamond A}(\diamond_{\mathcal{I}}) \qquad \frac{\Gamma \vdash \diamond A \qquad \Gamma, A \vdash \diamond B}{\Gamma \vdash \diamond B}(\diamond_{\mathcal{E}})$$

Interestingly, not only was (the Hilbert-style presentation of) this logic discovered by Fairtlough and Mendler (who call it 'lax logic') in the context of hardware verification [FM95], but it had even been considered by Curry in 1957 [Cur57]! Moreover, from a logical perspective, the three basic equations of the computational metalanguage arise as inevitable consequences of the form of the introduction and elimination rules, rather than being imposed separately.

The way in which the β-rule for the computation type constructor arises from the natural deduction presentation of the logic is fairly straightforward – the basic step in normalization is the removal of 'detours' caused by the introduction and immediate elimination of a logical connective:

$$\frac{\dfrac{\begin{array}{c}\vdots\\ A\end{array}}{\diamond A}(\diamond_{\mathcal{I}}) \qquad \begin{array}{c}[A]\cdots[A]\\ \vdots\\ \diamond B\end{array}}{\diamond B}(\diamond_{\mathcal{E}}) \quad \longrightarrow \quad \begin{array}{c}\vdots\\ [A]\end{array} \cdots \begin{array}{c}\vdots\\ [A]\end{array} \quad \Rightarrow \quad \begin{array}{c}\vdots\\ \diamond B\end{array}$$

$$\dfrac{\dfrac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{val}\, M : TA} \qquad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathrm{let}\ x \Leftarrow \mathrm{val}\, M\ \mathrm{in}\ N : TB} \qquad \longrightarrow \qquad \Gamma \vdash N[M/x] : TB$$

Natural deduction systems can also give rise to a secondary form of normalisation step which are necessary to ensure that normal deductions satisfy the subformula property, for example. These occur when the system contains elimination rules which have a minor premiss (Girard calls this a 'parasitic formula' and refers to the necessity for these extra reduction 'the shame of natural deduction' [GLT89]). In general, when we have such a rule, we want to be able to commute the last rule in the derivation of the minor premiss down past the rule, or to move the application of a rule to the conclusion of the elimination up past the elimination rule into to the derivation of the minor premiss. The only important cases are moving eliminations up or introductions down. Such transformations are called *commuting conversions*. The elimination rule for disjunction (coproducts) in intuitionisitic logic gives rise to commuting conversions and so does the elimination for the $\diamond$ modality of CL-logic. The restriction on the form of the conclusion of our $(\diamond_\mathcal{E})$ rule (it must be modal) means that the rule gives rise to only one commuting conversion:

– A deduction of the form

$$\dfrac{\dfrac{\diamond A \qquad \overset{[A]}{\vdots}\ \diamond B}{\diamond B}\,(\diamond_\mathcal{E}) \qquad \overset{[B]}{\vdots}\ \diamond C}{\diamond C}\,(\diamond_\mathcal{E})$$

commutes to

$$\dfrac{\diamond A \qquad \dfrac{\overset{[A]}{\vdots}\ \diamond B \qquad \overset{[B]}{\vdots}\ \diamond C}{\diamond C}\,(\diamond_\mathcal{E})}{\diamond C}\,(\diamond_\mathcal{E})$$

On terms of the computational metalanguage, this commuting conversion induces the 'let of a let' associativity rule:

$$\dfrac{\dfrac{\Gamma \vdash M : TA \qquad \Gamma, y : A \vdash P : TB}{\Gamma \vdash \mathrm{let}\ y \Leftarrow M\ \mathrm{in}\ P : TB} \qquad \Gamma, x : B \vdash N : TC}{\Gamma \vdash \mathrm{let}\ x \Leftarrow (\mathrm{let}\ y \Leftarrow M\ \mathrm{in}\ P)\ \mathrm{in}\ N : TC} \qquad \longrightarrow$$

$$\dfrac{\Gamma \vdash M : TA \qquad \dfrac{\Gamma, y : A \vdash P : TB \qquad \Gamma, y : A, x : B \vdash N : TC}{\Gamma, y : A \vdash \mathrm{let}\ x \Leftarrow P\ \mathrm{in}\ N : TC}}{\Gamma \vdash \mathrm{let}\ y \Leftarrow M\ \mathrm{in}\ (\mathrm{let}\ x \Leftarrow P\ \mathrm{in}\ N) : TC}$$

Commuting conversions are not generally optimizing tranformations in their own right, but they reorganise code so as to expose more computationally significant β reductions. They are therefore important in compilation, and most compilers for functional languages perform at least some of them. MLj is somewhat dogmatic in performing *all* of them, to reach what we call *cc-normal form*, from which it also turns out to be particularly straighforward to generate code. As Danvy and Hatcliff observe [HD94], this is closely related to working with A-normal forms, though the logical/proof theoretic notion is an older and more precisely defined pattern.

Acknowledgements

We have used Paul Taylor's package for diagrams.

References

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[BBdP98] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, March 1998. Preliminary version appeared as Technical Report 365, University of Cambridge Computer Laboratory, May 1995.

[Ben92] P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, Computer Laboratory, University of Cambridge, December 1992.

[BHR99] A. Banerjee, N. Heintze, and J. G. Reicke. Region analysis and the polymorphic lambda calculus. In *Fourteenth IEEE Symposium on Logic and Computer Science*, 1999.

[BHT98] G. Barthe, J. Hatcliff, and P. Thiemann. Monadic type systems: Pure type systems for impure settings. In *Proceedings of the Second HOOTS Workshop, Stanford University, Palo Alto, CA. December, 1997*, Electronic Notes in Theoretical Computer Science. Elsevier, February 1998.

[BK99] N. Benton and A. Kennedy. Monads, effects and transformations. In *Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), Paris*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999.

[BK01] N. Benton and A. Kennedy. Exceptional syntax. *Journal of Functional Programming*, 2001. To appear.

[BKR98] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN Conference on Functional Programming*, September 1998.

[BM92] G. L. Burn and D. Le Metayer. Proving the correctness of compiler optimisations based on a global program analysis. Technical Report Doc 92/20, Department of Computing, Imperial College, London, 1992.

[BNTW95] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1), 1995.

[Bor94] F. Borceux. *Handbook of Categorial Algebra.* Cambridge University Press, 1994.

[BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, pages 579–612, 1996.

[Bur75] W. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, Reading, Mass., 1975.

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories.* Springer, 1985.

[Cal01] C. Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2001.

[CM93] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *CTCS 1993*, 1993. CWI Tech. Report.

[Cur57] H. B. Curry. The elimination theorem when modality is present. *Journal of Symbolic Logic*, 17(4):249–265, January 1957.

[DH93] O. Danvy and J. Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3), 1993.

[dZG00] S. dal Zilio and A. Gordon. Region analysis and a π-calculus with groups. In *25th International Symposium on Mathematical Foundations of Computer Science*, August 2000.

[Fil99] A. Filinski. Representing layered monads. In *POPL'99*. ACM Press, 1999.

[FM95] M. Fairtlough and M. Mendler. An intuitionistic modal logic with applications to the formal verification of hardware. In *Proceedings of Computer Science Logic 1994*, volume 933 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[FSDF93] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation*. ACM, 1993.

[GJLS87] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, 1987.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*. ACM Press, 1986.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Gor79] M.J.C. Gordon. *Denotational Description of Programming Languages*. Springer, 1979.

[HD94] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*. ACM, January 1994.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[HT00] S. Helsen and P. Thiemann. Syntactic type soundness for the region calculus. In *Workshop on Higher Order Operational Techniques in Semantics*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2000.

[Hud98] P. Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, pages 134–142, Victoria, Canada, June 1998.

[Hug99] John Hughes. Restricted Datatypes in Haskell. In *Third Haskell Workshop*. Utrecht University technical report, 1999.

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, 1999.

[JHe+99] Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[JLST98] S. L. Peyton Jones, J. Launchbury, M. B. Shields, and A. P. Tolmach. Bridging the gulf: A common intermediate language for ML and Haskell. In *Proceedings of POPL'98*. ACM, 1998.

[Jon96] S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proceedings of the European Symposium on Programming, Linkping, Sweden*, number 1058 in Lecture Notes in Computer Science. Springer-Verlag, January 1996.

[JRtYHG+99] Mark P Jones, Alastair Reid, the Yale Haskell Group, the Oregon Graduate Institute of Science, and Technology. The hugs 98 user manual. Available from `http://haskell.org/hugs`, 1994-1999.

[JW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *20'th Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993. ACM Press.

[KKR+86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices, pages 219–233, 1986.

[KL95] D. King and J. Launchbury. Structuring depth-first search algorithms in haskell. In *Conf. Record 22nd Symp. on Principles of Programming Languages*, pages 344–354, San Francisco, California, 1995. ACM.

[Lev99] P.B. Levy. Call-by-push-value: a subsuming paradigm (extended abstract). In *Typed Lambda-Calculi and Applications*, volume 1581 of *LNCS*. Springer, 1999.

[LH96] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96*, volume 1058 of *LNCS*. Springer, 1996.

[LHJ95] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*. ACM Press, 1995.

[LJ94] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

[LS97] J. Launchbury and A. Sabry. Monadic state: Axiomatisation and type safety. In *Proceedings of the International Conference on Functional Programming*. ACM, 1997.

[LW91] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proceddings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, 1991.

[Man76] E. Manes. *Algebraic Theories*. Graduate Texts in Mathematics. Springer, 1976.

[Man98] E Manes. Implementing collection classes with monads. *Mathematical Structures in Computer Science*, 8(3), 1998.

[Min98] Y. Minamide. A functional represention of data structures with a hole. In *Proceedings of the 25rd Symposium on Principles of Programming Languages*, 1998.

[MMH96] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida*. ACM, January 1996.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asiloomar, CA*, pages 14–23, 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Mog95] E. Moggi. A semantics for evaluation logic. *Fundamenta Informaticae*, 22(1/2), 1995.

[Mog97] E. Moggi. Metalanguages and applications. In *Semantics and Logics of Computation*, volume 14 of *Publications of the Newton Institute*. Cambridge University Press, 1997.

[Mos90] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 11. MIT press, 1990.

[Mos92] P. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[MP99] E. Moggi and F. Palumbo. Monadic encapsulation of effects: A revised approach. In *Proceedings of the Third International Workshop on Higher-Order Operational Techniques in Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier, September 1999.

[MT91] I. Mason and C. Talcott. Equivalences in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[NHH99] F. Nielson, H. R. Hielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Pey01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In R Steinbrueggen, editor, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, NATO ASI Series. IOS Press, 2001.

[Pit97] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[Pit00a] A. M. Pitts. Operational semantics and program equivalence. revised version of lectures at the international summer school on applied semantics. This volume, September 2000.

[Pit00b] A.M. Pitts. Categorical logic. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5. Oxford University Press, 2000.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Sco93] D.S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Science*, 121, 1993.

[SF93] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[Sha97] Z. Shao. An overview of the FLINT/ML compiler. In *Proceedings of the 1997 ACM Workshop on Types in Compilation, Amsterdam*. ACM, June 1997.

[SS99] M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *Proceedings of the International Conference on Functional Programming*. ACM, 1999.

[Sta94] I. D. B. Stark. *Names and Higher Order Functions.* PhD thesis, Computer Laboratory, University of Cambridge, 1994.

[TJ94] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), June 1994. Revised from LICS 1992.

[Tof87] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Department of Computer Science, University of Edinburgh, 1987.

[Tol98] A. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Proceedings of the Workshop on Types in Compilation, Kyoto*, March 1998.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2), February 1997.

[Wad92] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2, 1992.

[Wad98] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming.* ACM Press, 1998.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[Wri95] A. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8:343–355, 1995.

[WT99] P. Wadler and P. Thiemann. The marriage of effects and monads. Submitted to ACM Transactions on Computational Logic, 1999.