A Monadic Categorical Semantics for Threads

Jonathan Sobel
Indiana University

Steven E. Ganz
Indiana University

Daniel P. Friedman
Indiana University

Mitchell Wand
Northeastern University

Abstract

Rather than relegating threads to be a machine-level concept, it is possible to describe thread semantics at the programming-language level. Here, a categorical semantics is given for a potentially multithreaded programming language. The thread behavior is packaged as a monad, and the semantics is described both directly and via a semantics-preserving "trampolining" translation into a λ-calculus-like metalanguage. The semantics captures the notions of incremental work, interleaving, and it supports language-level thread operators. Different thread monads, their properties, and the operators they support are discussed.

1 Introduction

If one turns one's attention to the machine level, then the basic operation of threads may be explained without much difficulty. Each thread has its own stack and registers—including, most importantly, its own instruction pointer. All threads share the heap space. Each thread gets to execute several instructions before its state (the stack and registers) is saved, and another thread becomes active.

If one's attention shifts, however, to the level of expressions in a programming language, then it is no longer clear how to explain threads. What is an "instruction"? How is the state of the computation captured and preserved? In particular, what corresponds to the machine-level's instruction pointer?

One way to respond to these questions is to avoid them, leaving threads as a machine-level concept only. The problem with this (non-)approach is that one often wants special thread operations, such as creating or suspending threads, to be available in the high-level language. Without an account of threads that fits the language level, there is no way to describe the semantics of these operations.

Ganz, Friedman, and Wand describe a programming style that supports multithreading in a non-threaded functional language [4]. Here, we offer a monadic categorical semantics for a programming language that supports threads and language-level thread operations. The following section describes the semantic and computational notions on which the semantics is based. It also includes the definition of a simple thread monad. Section 3 describes the programming language and its semantics. Section 4 introduces a metalanguage, which is *not* threaded, and defines a translation from the programming language into the metalanguage that is correct in its preservation of the thread semantics. This translation is based on the trampolining translation of Ganz, Friedman, and Wand. Section 5 explores the thread monad in more depth and shows how, by varying the definition of the monad, differing amounts of control over the threads and their concurrency may be obtained, and different thread operators may be supported by the programming language. The conclusion discusses some other related work and how it might be combined with this work to better link the language-level and machine-level explanations of threads.

2 Monadic Categorical Semantics and the Resumption Monad

The kernel of our technique for representing threaded computations is similar to Plotkin's notion of *resumptions* [8]. The basic idea is that a finished computation produces a value, but an unfinished one produces an object that can be used to restart the computation where it left off. This object must capture all the information needed to resume the computation, corresponding to the instruction pointer, stack, etc., in the machine-level characterization of threads. In a functional setting, zero-argument closures make good resumption objects.

As a concrete example, the expected type for computations that produce integer values is[1]

$$\mathit{IntThread} = \mathit{Int} + (1 \to \mathit{IntThread})$$

Thus, the computation may immediately produce an integer value, or it may require a single step (involving the invocation of a thunk) and then produce an integer value, and so on, recursively.

The standard way to define a categorical semantics is to interpret programs (well-formedness proofs, really) as arrows in a suitable category $\mathbf{C}$. Following Moggi's recommendation [7] for handling languages with some non-mathematical notion of computation (in our case, threads), we describe the "imperative" portion of the semantics using a monad in $\mathbf{C}$. The *IntThread* type may be generalized to construct a *resumption monad* in some Cartesian closed category with sums.[2] The simplest resumption monad $\langle T, \eta, \mu \rangle$ is defined as:

$$\begin{aligned} Tx &= x + (Tx)^1 && (1) \\ \eta_A &= \iota_1^{A,(TA)^1} && (2) \\ \mu_A &= (\mathit{1}_{TA} \mid \beta_A \circ (\mu_A)^1) && (3) \end{aligned}$$

where β is a natural transformation defined as:

$$\beta_A = \iota_2^{A,(TA)^1} \qquad (4)$$

(See Appendix A for a discussion of our mathematical notation.) Just as η injects values into the monad, β, which stands for *bounce* [4], injects frozen computations into the monad. It is used in the semantics and definitions of other operators, too. This monad is a variation of an interactive input monad [7] in which the input type is always the unit type. The proof that it is a monad is not included here.

[1]Throughout this paper, whenever we use recursive equations, we are implicitly referring to a least-fixed-point construction. For example, this equation abbreviates

$$\mathit{IntThread} = \mu x.\mathit{Int} + (1 \to x)$$

[2]We use the term *sums* advisedly. The feature that distinguishes sums from other coproducts is that they are required to satisfy the distributive law with products. This property is necessary for giving a categorical semantics for languages with sum types.

In the simple resumption monad, g^* (after some simplification) is:

$$g^* = (g \mid \beta_C \circ (g^*)^1) \tag{5}$$

This Kleisli star operator corresponds to a Curried version of a thread-sequencing operator [4].

Given the monad T defined over $\mathbf{C}$, we interpret programs, by induction on well-formedness proofs, as arrows in the Kleisli category $\mathbf{C}_T$ induced by the monad. A description of the Kleisli construction is given in Appendix B.

A program returning a value of type *Int* is interpreted as an arrow whose codomain is $[\![Int]\!]_T$ in $\mathbf{C}_T$, which corresponds to an arrow whose codomain is $T([\![Int]\!])$ in $\mathbf{C}$. Thus, what the programmer sees as a program producing an integer, we interpret as a program producing an integer-valued thread.

In order to treat the notion of "steps of work" more formally, we define a pair of related operations σ and $\boxed{f}$. First, we define the "step" operator $TA \xrightarrow{\sigma_A} TA$ as follows:

$$\sigma_A = (\eta_A \mid \rho_{TA}) \tag{6}$$

where the "resume" operator $A^1 \xrightarrow{\rho_A} A$ is defined as:

$$\rho_A = e_{1,A} \circ (\mathit{1}_{A^1} \,\&\, !_{A^1}) \tag{7}$$

The definition of ρ is general. It evaluates a terminal-object exponential (a thunk). The definition of σ is specific to the simple resumption monad. If the thread is finished, it does nothing; if the thread is unfinished, it resumes the computation.

Next, we define an operation on arrows $A \xrightarrow{f} TB$ that produces a new arrow with the same domain and codomain, but requiring "one more step" to complete its computation:

$$\boxed{f} = \beta_B \circ \ulcorner f \circ \pi_1^{A,1} \urcorner \tag{8}$$

The arrow being Curried is the composition

$$A \times 1 \xrightarrow{\pi_1^{A,1}} A \xrightarrow{f} TB$$

which throws away the right element of its input pair. The application of the Currying operator produces

$$A \xrightarrow{\ulcorner f \circ \pi_1^{A,1} \urcorner} TB^1$$

Finally, β injects this "frozen computation" back into the monad.

The intended "stepping" behavior of σ may be characterized by its treatment of finished and unfinished computations. The following equations capture this characterization:

$$\sigma_A \circ \eta_A = \eta_A \tag{9}$$
$$\sigma_B \circ \boxed{f} = f \tag{10}$$

where $A \xrightarrow{f} TB$. Thus, taking a step does nothing to a finished computation, but taking a step removes exactly one layer of delay from an unfinished computation. These equations, together with one that requires the naturality of β to extend to its interaction with μ (even though mu is not actually of the right form to benefit from β's naturality), form the constraints on β, σ, and $\boxed{f}$ that we require to hold of all thread monads, not just the resumption monad. The equation for β is:

$$\mu_A \circ \beta_{TA} = \beta_A \circ (\mu_A)^1 \tag{11}$$

Theorem 1 *The resumption monad satisfies Equations 9, 10, and 11.*

Proof. Using simple properties from the diagrams in Appendix A, plus the fact that $!_B \circ g = !_A$ for any $A \xrightarrow{g} B$ (by uniqueness of terminal arrows). The most complex is Equation 10, proven here:

$$\begin{aligned}
\sigma_B \circ \boxed{f} &= (\eta_B \mid e_{1,TB} \circ (1_{(TB)^1} \,\&\, !_{(TB)^1})) \circ \beta_B \circ \ulcorner f \circ \pi_1^{A,1} \urcorner \\
&= e_{1,TB} \circ (1_{(TB)^1} \,\&\, !_{(TB)^1}) \circ \ulcorner f \circ \pi_1^{A,1} \urcorner \\
&= e_{1,TB} \circ ((1_{(TB)^1} \circ \ulcorner f \circ \pi_1^{A,1} \urcorner) \,\&\, (!_{(TB)^1} \circ \ulcorner f \circ \pi_1^{A,1} \urcorner)) \\
&= e_{1,TB} \circ (\ulcorner f \circ \pi_1^{A,1} \urcorner \,\&\, !_A) \\
&= e_{1,TB} \circ (\ulcorner f \circ \pi_1^{A,1} \urcorner \times 1_1) \circ (1_A \,\&\, !_A) \\
&= f \circ \pi_1^{A,1} \circ (1_A \,\&\, !_A) \\
&= f \circ 1_A \\
&= f
\end{aligned}$$

A few details remain to be covered before we can define the semantics of the threaded programming language. First, it is not enough to have only a monad; languages that support multiple free variables in expressions require a *strong monad*, which is a monad with an additional transformation $(A \times TB) \xrightarrow{t_{A,B}} T(A \times B)$. This transformation must, like the rest of the monad, satisfy several equalities, in conjunction with projections, associativity, and combination with η and μ. We defer to other sources [7] for details. The effect of t is to mediate sequential compositions in which an extra value (the first item in the pair) is carried along unchanged. For the resumption monad, t is defined as follows (where $\llcorner f \lrcorner$ "un-Curries" f, as described in Appendix A):

$$t_{A,B} = \llcorner \lambda a : A \,.\, \mathrm{fix}(\lambda f : (TB \to T(A \times B)) \,.\, ((\lambda b : B \,.\, \eta_{A\times B}(\langle a, b\rangle)) \mid \beta_{A\times B} \circ f^1)) \lrcorner \tag{12}$$

Then, in order to handle the "merging" of threads—especially in pairing operations—we define a transformation $(TA \times TB) \xrightarrow{\psi_{A,B}} T(A \times B)$, in terms of t and the "commute" isomorphism $A \times B \xrightarrow{\kappa_{A,B}} B \times A$. The effect of ψ is to mediate sequential compositions involving product-forming arrows. The left element of the pair "evaluated" first, then the right element.

$$\psi_{A,B} = (t_{A,B} \circ \kappa_{TB,A})^* \circ t_{TB,A} \circ \kappa_{TA,TB} \tag{13}$$

3 A Threaded Programming Language

The syntax of the programming language appears in Table 1. Types of the form B are base types. The semantics of the language is parametric in the interpretation of the base types, which must denote objects $[\![\mathrm{B}]\!]$ in $\mathbf{C}$ (or, equivalently, objects $[\![\mathrm{B}]\!]_T$ in $\mathbf{C}_T$). Terms of the form $\mathrm{p}(E)$ represent

E_{pl}	$::=$	$x \mid \mathrm{p}(E_{pl}) \mid \mathrm{let}\ x = E_{pl}\ \mathrm{in}\ E_{pl}$
		$\mid * \mid \langle E_{pl}, E_{pl}\rangle \mid \mathrm{fst}(E_{pl}) \mid \mathrm{snd}(E_{pl})$
		$\mid \lambda x : \tau_{pl} \,.\, E_{pl} \mid E_{pl}(E_{pl})$
		$\mid \mathrm{inl}(E_{pl}) \mid \mathrm{inr}(E_{pl}) \mid \mathrm{sumcase}\ E_{pl}\ \mathrm{or}\ E_{pl}$
		$\mid [E_{pl}] \mid \mu(E_{pl})$
τ_{pl}	$::=$	$\mathrm{B} \mid 1 \mid \tau_{pl} \times \tau_{pl} \mid \tau_{pl} \to \tau_{pl} \mid \tau_{pl} + \tau_{pl} \mid T(\tau_{pl})$

Table 1: Term and Type Syntax for the Programming Language

calls to primitive commands $\mathrm{p} : \tau_1 \rightharpoonup \tau_2$. The semantics of the language is also parametric in the interpretation of primitive commands, but each must denote some arrow $[\![\tau_1]\!] \xrightarrow{[\![\mathrm{p}]\!]} T([\![\tau_2]\!])$ in $\mathbf{C}$ (or, equivalently, some arrow $[\![\tau_1]\!] \xrightarrow{[\![\mathrm{p}]\!]^\flat} [\![\tau_2]\!]$ in $\mathbf{C}_T$).

The "sumcase" form is not one we have seen elsewhere, but it arises naturally from including categorical sums in the semantics. In words, it requires the most recently bound variable (i.e., lexically innermost) to be of a sum type $\tau_1 + \tau_2$. If its value is left-injected, the value of the "sumcase" form is the value of its first subexpression, in which the variable is treated as if it were simply of type τ_1; if the value is right-injected, the value of the form is that of the second subexpression, in which the variable is treated as if it were of type τ_2. Perhaps in a more programmer-friendly language, one would define a "type-case" form (taking an explicit argument on which to dispatch) in terms of "sumcase."

The "let" form is technically unnecessary here, as it could be defined in terms of λ and application, but it is included for completeness and because it is more primitive than λ. The last two forms provide monadic reification and reflection, respectively [7, 2, 3]. In Section 5, we use them to define more specific thread-related operations.

The semantics for the various type constructors (and environments) is given in Table 2. They are all standard for monadic categorical semantics, although the meaning of arrow types is different from the usual denotation in non-monadic settings. Technically, all the objects described in the type semantics are in the Kleisli category $\mathbf{C}_T$, and so they should all have subscripts, as in $(A_1 \times A_2)_T$. For the sake of brevity and clarity—and because $\mathbf{C}$ and $\mathbf{C}_T$ contain the same objects—these have been omitted. (They become illegible when the objects themselves appear as subscripts in the term semantics.)

The semantics of the programming language is defined in Tables 3 and 4. As defined, the language is call-by-value. The semantics of "snd" and "inr" have been omitted, since the are so nearly identical to "fst" and "inl," respectively.

There are two important places where this semantics differs from what is typical: the rule for primitive commands and the rule for application. In both places, the $\boxed{f}$ operation causes the computation to require an additional unit of work. For instance, the program

$$x : \mathit{Int} \vdash \mathrm{add1}(\mathrm{add1}(x)) \ : \ \mathit{Int}$$

denotes $\mathit{Int} \xrightarrow{add2} \mathit{Int}$ in a regular categorical semantics. Here, the same program denotes an arrow

B			
	$\vdash \mathrm{B}$ type	$=$	$[\![\mathrm{B}]\!]$
1			
	$\vdash 1$ type	$=$	1
$\times$			
	$\vdash \tau_1$ type	$=$	A_1
	$\vdash \tau_2$ type	$=$	A_2
	$\vdash \tau_1 \times \tau_2$ type	$=$	$A_1 \times A_2$
$\rightarrow$			
	$\vdash \tau_1$ type	$=$	A_1
	$\vdash \tau_2$ type	$=$	A_2
	$\vdash \tau_1 \rightarrow \tau_2$ type	$=$	$(TA_2)^{A_1}$
$+$			
	$\vdash \tau_1$ type	$=$	A_1
	$\vdash \tau_2$ type	$=$	A_2
	$\vdash \tau_1 + \tau_2$ type	$=$	$A_1 + A_2$
T			
	$\vdash \tau$ type	$=$	A
	$\vdash T(\tau)$ type	$=$	TA
env			
	$\vdash \tau_i$ type $(1 \leq i \leq n)$	$=$	A_i
	$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash$	$=$	$A_1 \times \cdots \times A_n$

Table 2: Interpretation of Types in the Programming Language

var_i	$\begin{array}{l} \vdash \tau_j \text{ type } (1 \leq j \leq n) \\ \hline x_1 : \tau_1, \dots, x_n : \tau_n \vdash x_i \ : \ \tau_i \end{array}$	$\begin{array}{l} = \ A_j \\ = \ F_T(\pi_i^{A_1, \dots, A_n}) \end{array}$
$\mathrm{p} : \tau_1 \rightharpoonup \tau_2$	$\begin{array}{l} \Gamma \vdash E \ : \ \tau_1 \\ \hline \Gamma \vdash \mathrm{p}(E) \ : \ \tau_2 \end{array}$	$\begin{array}{l} = \ g^\flat \\ = \ \boxed{\llbracket \mathrm{p} \rrbracket}^\flat \circ g^\flat \end{array}$
$*$	$\begin{array}{l} \\ \hline \Gamma \vdash * \ : \ 1 \end{array}$	$= \ F_T(!_{\llbracket \Gamma \rrbracket})$
$\langle , \rangle$	$\begin{array}{l} \Gamma \vdash E_1 \ : \ \tau_1 \\ \Gamma \vdash E_2 \ : \ \tau_2 \\ \hline \Gamma \vdash \langle E_1, E_2 \rangle \ : \ \tau_1 \times \tau_2 \end{array}$	$\begin{array}{l} = \ g_1^\flat \\ = \ g_2^\flat \\ = \ (\psi_{\llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket} \circ (g_1 \,\&\, g_2))^\flat \end{array}$
fst	$\begin{array}{l} \Gamma \vdash E \ : \ \tau_1 \times \tau_2 \\ \hline \Gamma \vdash \mathrm{fst}(E) \ : \ \tau_1 \end{array}$	$\begin{array}{l} = \ g^\flat \\ = \ F_T(\pi_1^{\llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket}) \circ g^\flat \end{array}$
λ	$\begin{array}{l} \Gamma, x : \tau_1 \vdash E \ : \ \tau_2 \\ \hline \Gamma \vdash \lambda x : \tau_1 \,.\, E \ : \ \tau_1 \to \tau_2 \end{array}$	$\begin{array}{l} = \ g^\flat \\ = \ F_T(\ulcorner g \urcorner) \end{array}$
app	$\begin{array}{l} \Gamma \vdash E_1 \ : \ \tau_1 \\ \Gamma \vdash E \ : \ \tau_1 \to \tau_2 \\ \hline \Gamma \vdash E(E_1) \ : \ \tau_2 \end{array}$	$\begin{array}{l} = \ g_1^\flat \\ = \ g^\flat \\ = \ \boxed{e_{\llbracket \tau_1 \rrbracket, \llbracket T\tau_2 \rrbracket}}^\flat \\ \quad \circ (\psi_{\llbracket (T\tau_2)^{\tau_1} \rrbracket, \llbracket \tau_1 \rrbracket} \circ (g \,\&\, g_1))^\flat \end{array}$

Table 3: Interpretation of Terms in the Programming Language (Part 1)

inl	$\begin{array}{lcl} \Gamma \vdash E \ : \ \tau_1 & = & g^\flat \\ \hline \Gamma \vdash \mathrm{inl}(E) \ : \ \tau_1 + \tau_2 & = & F_T(\iota_1^{\llbracket\tau_1\rrbracket,\llbracket\tau_2\rrbracket}) \circ g^\flat \end{array}$
sumcase or	$\begin{array}{lcl} \Gamma, x : \tau_1 \vdash E_1 \ : \ \tau & = & g_1^\flat \\ \Gamma, x : \tau_2 \vdash E_2 \ : \ \tau & = & g_2^\flat \\ \hline \Gamma, x : (\tau_1 + \tau_2) \vdash \mathrm{sumcase}\ E_1 \ \mathrm{or}\ E_2 \ : \ \tau & = & ((g_1 \mid g_2) \circ \delta_{\llbracket\Gamma\rrbracket,\llbracket\tau_1\rrbracket,\llbracket\tau_2\rrbracket}{}^{-1})^\flat \end{array}$
let	$\begin{array}{lcl} \Gamma \vdash E_1 \ : \ \tau_1 & = & g_1^\flat \\ \Gamma, x : \tau_1 \vdash E_2 \ : \ \tau_2 & = & g_2^\flat \\ \hline \Gamma \vdash \mathrm{let}\ x = E_1 \ \mathrm{in}\ E_2 \ : \ \tau_2 & = & g_2^\flat \circ (t_{\llbracket\Gamma\rrbracket,\llbracket\tau_1\rrbracket} \circ (1_{\llbracket\Gamma\rrbracket} \,\&\, g_1))^\flat \end{array}$
$[-]$	$\begin{array}{lcl} \Gamma \vdash E \ : \ \tau & = & g^\flat \\ \hline \Gamma \vdash [E] \ : \ T(\tau) & = & F_T(\eta_{\llbracket\tau\rrbracket}) \circ g^\flat \end{array}$
$\mu(-)$	$\begin{array}{lcl} \Gamma \vdash E \ : \ T(\tau) & = & g^\flat \\ \hline \Gamma \vdash \mu(E) \ : \ \tau & = & (\varepsilon_T)_{\llbracket\tau\rrbracket_T} \circ g^\flat \end{array}$

Table 4: Interpretation of Terms in the Programming Language (Part 2)

which is actually $Int \xrightarrow{add2'} T(Int)$, but which is effectively

$$Int \xrightarrow{add2'} (Int^1)^1$$

except that the exponentiation is hidden by right injections. The same sort of "delay" occurs in the application rule, where the transformation is applied directly to the evaluation map e.

Under this semantics, the language has some unusual properties that make it fundamentally different from most mathematically defined languages. For instance, suppose the language has primitives "two," "three," and "five," each of type $1 \to Int$. Furthermore, suppose the arrows they denote each pick out the obvious left-injected integer values. If the language also has "+" as a primitive (taking a pair of *Int*s and returning a single one), then it is *not* true that $2+3=5$. That is, the programs

$$+(\langle \mathrm{two}(*), \mathrm{three}(*) \rangle)$$

and

$$\mathrm{five}(*)$$

do not have identical meanings.

Have we constructed a broken language, where nothing is as it seems? No, not at all. In any real language the computation produced by $2+3$ and the computation produced by 5 are different,[3] in that they take a different number of steps to compute their result. While this could be expressed indirectly in a reduction semantics, it is significant that the distinction may also be preserved in a categorical setting. Furthermore, it is still possible to define a notion of *equivalence* between $2+3$ and 5. The idea is that after some number of steps, the values of the two computations converge.

More precisely, say that, given denotations f and g,

$$f \cong_\sigma g \Longleftrightarrow \overbrace{\sigma \circ \cdots \circ \sigma \circ f}^{n} = \overbrace{\sigma \circ \cdots \circ \sigma \circ g}^{m} \tag{14}$$

for some finite m and n.

4 A Non-Threaded Metalanguage

In this section, we define a functional metalanguage (a typed λ-calculus with sums and products) with monadic operations for strict sequentialization and for injecting values and delayed computations into monadic types. Then, we give a second semantics for the programming language, this time in the form of a translation into the metalanguage. The two versions of the semantics are equivalent.

The syntax of the metalanguage appears in Table 5. The core syntax (including all the types) of the metalanguage is the same as that of the programming language. The exceptions are the three forms for interacting with values of type $T(\_)$. The $[\_]_T$ form applies η to its argument. The "let_T" form introduces strictness and sequentialization into the metalanguage. (It could *not* be implemented in terms of λ and application.) Both of these forms are standard in metalanguages for

[3] We are, of course, assuming no compile-time constant folding here.

$$
\begin{array}{lcl}
E_{ml} & ::= & x \mid \mathrm{f}(E_{ml}) \\
 & & \mid * \mid \langle E_{ml}, E_{ml} \rangle \mid \mathrm{fst}(E_{ml}) \mid \mathrm{snd}(E_{ml}) \\
 & & \mid \lambda x : \tau_{ml} \,.\, E_{ml} \mid E_{ml}(E_{ml}) \\
 & & \mid \mathrm{inl}(E_{ml}) \mid \mathrm{inr}(E_{ml}) \mid \mathrm{sumcase}\ E_{ml}\ \mathrm{or}\ E_{ml} \\
 & & \mid [E_{ml}]_T \mid \rceil E_{ml} \lceil_T \mid \mathrm{let}_T\ x \Leftarrow E_{ml}\ \mathrm{in}\ E_{ml} \\
 & & \\
\tau_{ml} & ::= & \mathrm{B} \mid 1 \mid \tau_{ml} \times \tau_{ml} \mid \tau_{ml} \to \tau_{ml} \mid \tau_{ml} + \tau_{ml} \mid T(\tau_{ml})
\end{array}
$$

Table 5: Term and Type Syntax for the Metalanguage

monadic semantics. The $\rceil \_ \lceil_T$ form is particular to our metalanguage for threads; it applies β (from Equation 4) to its argument.

The metalanguage is interpreted directly into $\mathbf{C}$, not $\mathbf{C}_T$. As in the programming language, the semantics is parametric in the interpretation of base types and primitives. Terms of the form $\mathrm{f}(E)$ represent calls to primitive functions $\mathrm{f} : \tau_1 \to \tau_2$; they must denote arrows $[\![\tau_1]\!] \xrightarrow{[\![\mathrm{f}]\!]} [\![\tau_2]\!]$ in $\mathbf{C}$.

The semantics for the type constructors, given in Table 6, is completely standard this time. Arrow types are interpreted as ordinary exponentials, as in any standard λ-calculus.

The semantics of terms, given in Tables 7 and 8, is also standard. The rules for pairing and application, for example, are much simpler without the burden imposed by coping with an imperative language. Only the rule for "let_T" retains an interpretation involving the use of t, as expected.

Now, instead of interpreting the programming language directly, we can translate programs into terms of the metalanguage, and interpret the result. In order for this translation to make sense, the languages must have the same base types, and the metalanguage must have a primitive function $\mathrm{p} : \tau_1 \to T(\tau_2)$ for every primitive command $\mathrm{p} : \tau_1 \to \tau_2$ in the programming language. The translation is given in Table 9. The translation is correct in that the interpretation of translated programs in the metalanguage matches their original interpretation:

Theorem 2 *Every well-formed program $\Gamma \vdash E : \tau$ in the programming language can be translated into a well-formed term $\Gamma \vdash \overline{E} : T(\tau)$, and*

$$\varphi_T([\![\Gamma \vdash E : \tau]\!]_{pl}) = [\![\Gamma \vdash \overline{E} : T(\tau)]\!]_{ml}$$

Proof. By induction on the syntax trees of the programming language. Some of the cases are quite simple. For example, the semantics of the translation of variables is directly equal to their semantics in the programming language, with no additional simplification. Likewise for λ-terms. The programs that translate into a use of "let_T" generally require that the use of the Kleisli "star" operator $\_^*$ be expanded using its definition. Then, properties of t from the definition of strong monads must be invoked to eliminate the t (usually in conjunction with π_2).

5 More Thread Monads

The simple resumption monad supports only limited thread functionality. For example, suppose both languages have primitives for manipulating integers, as well as a family of "fix" primitives for

B

$$\begin{array}{lcl} \hline \vdash \mathrm{B}\ \text{type} & = & [\![\mathrm{B}]\!] \end{array}$$

1

$$\begin{array}{lcl} \hline \vdash 1\ \text{type} & = & 1 \end{array}$$

$\times$

$$\begin{array}{lcl} \vdash \tau_1\ \text{type} & = & A_1 \\ \vdash \tau_2\ \text{type} & = & A_2 \\ \hline \vdash \tau_1 \times \tau_2\ \text{type} & = & A_1 \times A_2 \end{array}$$

$\to$

$$\begin{array}{lcl} \vdash \tau_1\ \text{type} & = & A_1 \\ \vdash \tau_2\ \text{type} & = & A_2 \\ \hline \vdash \tau_1 \to \tau_2\ \text{type} & = & {A_2}^{A_1} \end{array}$$

$+$

$$\begin{array}{lcl} \vdash \tau_1\ \text{type} & = & A_1 \\ \vdash \tau_2\ \text{type} & = & A_2 \\ \hline \vdash \tau_1 + \tau_2\ \text{type} & = & A_1 + A_2 \end{array}$$

T

$$\begin{array}{lcl} \vdash \tau\ \text{type} & = & A \\ \hline \vdash T(\tau)\ \text{type} & = & TA \end{array}$$

env

$$\begin{array}{lcl} \vdash \tau_i\ \text{type}\ (1 \leq i \leq n) & = & A_i \\ \hline x_1 : \tau_1, \ldots, x_n : \tau_n \vdash & = & A_1 \times \cdots \times A_n \end{array}$$

Table 6: Interpretation of Types in the Metalanguage

var_i	$\begin{array}{l} \vdash \tau_j \text{ type } (1 \le j \le n) \\ \hline x_1 : \tau_1, \ldots , x_n : \tau_n \vdash x_i \ : \ \tau_i \end{array}$	$\begin{array}{l} = \ A_j \\ = \ \pi_i^{A_1, \ldots , A_n} \end{array}$
$\mathrm{f} : \tau_1 \to \tau_2$	$\begin{array}{l} \Gamma \vdash E \ : \ \tau_1 \\ \hline \Gamma \vdash \mathrm{f}(E) \ : \ \tau_2 \end{array}$	$\begin{array}{l} = \ g \\ = \ [\![\mathrm{f}]\!] \circ g \end{array}$
$*$	$\begin{array}{l} \\ \hline \Gamma \vdash * \ : \ 1 \end{array}$	$\begin{array}{l} \\ = \ !_{[\![\Gamma]\!]} \end{array}$
$\langle , \rangle$	$\begin{array}{l} \Gamma \vdash E_1 \ : \ \tau_1 \\ \Gamma \vdash E_2 \ : \ \tau_2 \\ \hline \Gamma \vdash \langle E_1, E_2 \rangle \ : \ \tau_1 \times \tau_2 \end{array}$	$\begin{array}{l} = \ g_1 \\ = \ g_2 \\ = \ (g_1 \ \& \ g_2) \end{array}$
fst	$\begin{array}{l} \Gamma \vdash E \ : \ \tau_1 \times \tau_2 \\ \hline \Gamma \vdash \mathrm{fst}(E) \ : \ \tau_1 \end{array}$	$\begin{array}{l} = \ g \\ = \ \pi_1^{[\![\tau_1]\!], [\![\tau_2]\!]} \circ g \end{array}$
λ	$\begin{array}{l} \Gamma, x : \tau_1 \vdash E \ : \ \tau_2 \\ \hline \Gamma \vdash \lambda x : \tau_1 . E \ : \ \tau_1 \to \tau_2 \end{array}$	$\begin{array}{l} = \ g \\ = \ \ulcorner g \urcorner \end{array}$
app	$\begin{array}{l} \Gamma \vdash E_1 \ : \ \tau_1 \\ \Gamma \vdash E \ : \ \tau_1 \to \tau_2 \\ \hline \Gamma \vdash E(E_1) \ : \ \tau_2 \end{array}$	$\begin{array}{l} = \ g_1 \\ = \ g \\ = \ e_{[\![\tau_1]\!], [\![\tau_2]\!]} \circ (g \ \& \ g_1)) \end{array}$

Table 7: Interpretation of Terms in the Metalanguage (Part 1)

inl	$\begin{array}{lcl} \Gamma \vdash E \;:\; \tau_1 & = & g \\ \hline \Gamma \vdash \mathrm{inl}(E) \;:\; \tau_1 + \tau_2 & = & \iota_1^{[\![\tau_1]\!],[\![\tau_2]\!]} \circ g \end{array}$
sumcase or	$\begin{array}{lcl} \Gamma, x : \tau_1 \vdash E_1 \;:\; \tau & = & g_1 \\ \Gamma, x : \tau_2 \vdash E_2 \;:\; \tau & = & g_2 \\ \hline \Gamma, x : (\tau_1 + \tau_2) \vdash \mathrm{sumcase}\ E_1\ \mathrm{or}\ E_2 \;:\; \tau & = & (g_1 \mid g_2) \circ {\delta_{[\![\Gamma]\!],[\![\tau_1]\!],[\![\tau_2]\!]}}^{-1} \end{array}$
$[\_]_T$	$\begin{array}{lcl} \Gamma \vdash E \;:\; \tau & = & g \\ \hline \Gamma \vdash [E]_T \;:\; T(\tau) & = & \eta_{[\![\tau]\!]} \circ g \end{array}$
$\rceil\_\lceil_T$	$\begin{array}{lcl} \Gamma \vdash E \;:\; 1 \to T(\tau) & = & g \\ \hline \Gamma \vdash \rceil E \lceil_T : T(\tau) & = & \beta_{[\![\tau]\!]} \circ g \end{array}$
let_T	$\begin{array}{lcl} \Gamma \vdash E_1 \;:\; T(\tau_1) & = & g_1 \\ \Gamma, x : \tau_1 \vdash E_2 \;:\; T(\tau_2) & = & g_2 \\ \hline \Gamma \vdash \mathrm{let}_T\ x \Leftarrow E_1\ \mathrm{in}\ E_2 \;:\; T(\tau_2) & = & g_2^* \circ t_{[\![\Gamma]\!],[\![\tau_1]\!]} \circ (1_{[\![\Gamma]\!]} \,\&\, g_1) \end{array}$

Table 8: Interpretation of Terms in the Metalanguage (Part 2)

$$
\begin{array}{rcl}
\overline{x} & = & [x]_T \\
\overline{\mathrm{p}(E)} & = & \mathrm{let}_T\ x \Leftarrow \overline{E}\ \mathrm{in}\ \rceil(\lambda v : 1\ .\ \mathrm{p}(x))\lceil_T \\
\overline{\mathrm{let}\ x = E_1\ \mathrm{in}\ E_2} & = & \mathrm{let}_T\ x \Leftarrow \overline{E_1}\ \mathrm{in}\ \overline{E_2} \\
\overline{*} & = & [*]_T \\
\overline{\langle E_1, E_2 \rangle} & = & \mathrm{let}_T\ x_1 \Leftarrow \overline{E_1}\ \mathrm{in}\ (\mathrm{let}_T\ x_2 \Leftarrow \overline{E_2}\ \mathrm{in}\ [\langle x_1, x_2 \rangle]_T) \\
\overline{\mathrm{fst}(E)} & = & \mathrm{let}_T\ x \Leftarrow \overline{E}\ \mathrm{in}\ [\mathrm{fst}(x)]_T \\
\overline{\lambda x : \tau\ .\ E} & = & [\lambda x : \tau\ .\ \overline{E}]_T \\
\overline{E_1(E_2)} & = & \mathrm{let}_T\ x_1 \Leftarrow \overline{E_1}\ \mathrm{in}\ (\mathrm{let}_T\ x_2 \Leftarrow \overline{E_2}\ \mathrm{in}\ \rceil(\lambda v : 1\ .\ x_1(x_2))\lceil_T) \\
\overline{\mathrm{inl}(E)} & = & \mathrm{let}_T\ x \Leftarrow \overline{E}\ \mathrm{in}\ [\mathrm{inl}(x)]_T \\
\overline{\mathrm{sumcase}\ E_1\ \mathrm{or}\ E_2} & = & \mathrm{sumcase}\ \overline{E_1}\ \mathrm{or}\ \overline{E_2} \\
\overline{[E]} & = & [\overline{E}]_T \\
\overline{\mu(E)} & = & \mathrm{let}_T\ x \Leftarrow \overline{E}\ \mathrm{in}\ x
\end{array}
$$

Table 9: Translation of the Programming Language into the Metalanguage

creating recursive programs. Then, the factorial function may be defined as:

$$
\begin{aligned}
\text{fact} = \text{fix}(\lambda f{:}(\mathit{Int}{\to}\mathit{Int}).\\
&\lambda n{:}\mathit{Int}.\\
&\ \ \text{let } z = \text{zero?}(n)\\
&\ \ \text{in sumcase one}(*) \text{ or } \times(\langle n, f(\text{sub1}(n))\rangle))
\end{aligned}
$$

The "sumcase" operator dispatches based on the value of the most recently bound variable, here z. Now, what does fact(five($*$)) denote, in an empty environment? In a standard categorical semantics, it denotes an arrow $1 \xrightarrow{120} \mathit{Int}$; i.e., it is interpreted as the point 120. In the semantics of this programming language, however, it denotes an arrow (which, when viewed in $\mathbf{C}$ is):

$$1 \xrightarrow{willbe120} T(\mathit{Int})$$

But how do we get the answer out?

Suppose the "step" operator σ is present as a primitive function (a family of them, really—one for each type) in the metalanguage. Then, it is possible to define "pogo"[4] in terms of σ:

$$\text{pogo}_A = \text{fix}(\lambda f : (TA \to A) \,.\, \lambda x : TA \,.\, (\lambda y : TA \,.\, \text{sumcase } y \text{ or } f(\sigma_A(x)))(x))$$

Thus, $\text{pogo}_{\mathit{Int}}(\text{fact}(\text{five}(*)))$ really is 120.[5]

While "pogo" demonstrates the little-bit-at-time nature of threads, it does not demonstrate any interleaving. This, too, is possible. The operator "seesaw," which takes a pair of threads and applies σ to each alternately, is also easy to define [4]. Even with the simple resumption monad, "seesaw" makes it possible to run fact(-1) and fact(5) simultaneously and get an answer.

To achieve more sophisticated multithreading, the thread monad must be redefined. In other words, the notion of computation represented by the monad must be enriched. Instead of representing a single thread, the next monad represents a thread queue. First, we need some operators from the list monad. Let *List* be the evident functor for the list monad. Let $\langle\rangle$ be the unit of the list monad, which maps values a to singleton lists $\langle a\rangle$. Finally, let Θ be the Kleisli "star" operator of the list monad, such that

$$\Theta f : \langle a_1, \ldots, a_n\rangle \mapsto f(a_1) \diamond \cdots \diamond f(a_n)$$

where $\diamond$ stands for list concatenation. Then the thread-queue monad is defined as:

$$
\begin{aligned}
Tx &= \mathit{List}(x + (Tx)^1) && (15)\\
\eta_A &= \langle\rangle_{A+(TA)^1} \circ \iota_1^{A,(TA)^1} && (16)\\
\mu_A &= \Theta(\mathit{1}_{TA} \mid \beta_A \circ (\mu_A)^1) && (17)
\end{aligned}
$$

where

$$\beta_A = \langle\rangle_{A+(TA)^1} \circ \iota_2^{A,(TA)^1} \tag{18}$$

[4] It bounces up and down like a pogo stick.

[5] There is an unresolved typed issue lurking here. See the following note on "die" and "spawn."

The definition of t, like that of μ, requires the addition of Θ:

$$t_{A,B} = \llcorner\lambda a : A \,.\, \mathrm{fix}(\lambda f : (TB \to T(A \times B)) \,.\, \Theta((\lambda b : B \,.\, \eta_{A\times B}(\langle a, b\rangle)) \mid \beta_{A\times B} \circ f^1))\lrcorner \tag{19}$$

With the new monad, the semantics of fact(5) is hardly changed. Instead of returning an element of a sum, it produces a singleton list containing an element of a sum. In fact, without the use of $[-]$ and $\mu(-)$ in the programming language, any computation will produce a singleton list. Thanks to the greater expressivity of the thread-queue monad, though, it is possible to define new thread operators in the programming language. If the language has primitive commands for manipulating lists, then programs may construct new thread queues.[6] For example, the following operator, defined in the programming language, causes a thread to "die" (i.e., be removed from the queue):

$$\mathrm{die} = \mu(\mathrm{emptylist}(*))$$

Programs can create new threads, too. The following operator, also defined in the programming language, causes two threads to take the place of one in the queue:

$$\mathrm{spawn}(E_1, E_2) = \mu(\mathrm{append}(\langle [E_1], [E_2]\rangle))$$

Whereas, with only the resumption monad, it requires a metalanguage operator (such as "seesaw") to run multiple computations simultaneously, with the thread-queue monad, it is possible to have

$$\mathrm{spawn}(\mathrm{fact}(-1), \mathrm{fact}(5))$$

as a single program.

In order to "run" these programs, a new σ is required. A simple definition yields a system in which all threads take a step forward in lock-step (if they have work to do):

$$\sigma_A = \Theta(\eta_A \mid \rho_{TA}) \tag{20}$$

It is also possible to define alternate versions of σ, allowing only one thread to make progress at each step, for instance.

6 Conclusion

One of the motivations for this work is to describe thread semantics at the programming-language level, not just at the machine level. The semantics given, both directly and via translation, is adequate for capturing the notions of incremental work, interleaving, and the creation and removal of threads (using an appropriate thread monad).

One issue this work does not address is the relationship between the language- and machine-level characterizations of threads. We believe that the relationship may be characterized by the application of linear types to the thread semantics. For example, it seems to be the case in the presence of all reasonable thread operators that no thread object is "stepped" twice. Taking a step creates a new thread value (or values), and the old one may be discarded. If this is true, then the thread objects may be reused, and they become strikingly similar to the register set that holds the instruction pointer, stack, etc., of each thread in the machine-level characterization. Taking this idea even farther, it seems that, using techniques like those described by Chen and Hudak [1], it should be possible to describe a threaded language monadically, as we do here, but implement it with mutation.

[6] Actually, the programming language, as it stands, has no means of converting into and out of $T(\_)$ types. To be technically correct, we should use *hide* and *reveal* operators to convert between, for example, TA and $List(A+(TA)^1)$.

References

[1] Chih-Ping Chen and Paul Hudak. Rolling your own mutable ADT: A connection between linear types and monads. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Paris, France, January 1997. ACM Press.

[2] Andrzej Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, New York, January 1994. ACM Press.

[3] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1996.

[4] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming '99*, Paris, September 1999. ACM Press. To appear.

[5] H. Kleisli. Every standard construction is induced by a pair of adjoint functors. In *Proceedings of the American Mathematical Society*, volume 16, pages 544–546, 1965.

[6] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, second edition, 1998.

[7] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[8] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, September 1976.

Appendices

A Mathematical Notation

In case some of the notation we use in this work is unfamiliar or different from what other authors have used, we describe it here. Certain type-styles and metavariables are used consistently throughout:

$\mathbf{C}$, $\mathbf{D}$	categories
A, B, C	objects in a category
1	terminal object
B^A	exponential object
f, g, h	arrows in a category
$\mathit{1}_A$	identity arrow for A
f^A	arrow between exponentials
$!_A$	unique arrow from A to 1
F, G, T	functors (or type constructors)
η	natural transformation (unit for monad)
μ	natural transformation (multiplication for monad)

The rest of the notation is summarized in the following commuting diagrams.

- Products:

$$\begin{tikzcd} C \arrow[r, equal] \arrow[d, "f"'] & C \arrow[r, equal] \arrow[d, "(f \,\&\, g)"] & C \arrow[d, "g"] \\ A & A \times B \arrow[l, "\pi_1^{A,B}"] \arrow[r, "\pi_2^{A,B}"'] & B \end{tikzcd}$$

  In our notation, composition is given higher precedence than the product-forming operator in expressions like $(f \,\&\, g \circ h)$.

  We denote by κ the natural isomorphism between $A \times B$ and $B \times A$:

$$A \times B \xrightarrow{\kappa_{A,B}} B \times A$$

  In the semantics of environments and variable references, we use extended products of the form $A \times B \times C$. This is an abbreviation for the left-associated, terminal-object-initiated product $((1 \times A) \times B) \times C$. In this context, the projections $\pi_i^{A,B,C}$ stand for the compositions of the appropriate simple projections.

- Coproducts:

$$\begin{tikzcd} A \arrow[r, "\iota_1^{A,B}"] \arrow[d, "f"'] & A + B \arrow[d, "(f \mid g)"] & B \arrow[l, "\iota_2^{A,B}"'] \arrow[d, "g"] \\ C \arrow[r, equal] & C \arrow[r, equal] & C \end{tikzcd}$$

  In our notation, composition is given higher precedence than the sum-casing operator in expressions like $(f \mid g \circ h)$.

- Distribution:

$$\begin{tikzcd} (A \times B) + (A \times C) \arrow[d, "\delta_{A,B,C}"] \\ A \times (B + C) \end{tikzcd}$$

  In *distributive categories*, δ is an isomorphism (thus, has an inverse), and we call the coproducts *sums*.

- Exponentials:

$$\begin{tikzcd} A \times B \arrow[r, "f"] \arrow[d, "(\ulcorner f \urcorner \times 1_B)"'] & C \arrow[d, equal] \\ C^B \times B \arrow[r, "e_{B,C}"'] & C \end{tikzcd}$$

  Notice that the "Currying" operation on f is denoted by $\ulcorner f \urcorner$. We denote its inverse ("un-Currying") by $\llcorner f \lrcorner$. Also, if $B \xrightarrow{g} C$, then $B^A \xrightarrow{g^A} C^A$ is defined as:

$$g^A = \lambda f : (A \to B) \,.\, g \circ f$$

B The Kleisli Construction

A monad might be the result of many different adjunctions. The adjunction given by the Kleisli construction [5, 6] is the "least" among them (i.e., initial in the category of all such adjunctions). Given the category $\mathbf{C}$ with the monad $\langle T, \eta, \mu \rangle$, the Kleisli category $\mathbf{C}_T$ is defined as follows:

- $\mathbf{C}_T$ contains an object A_T for every object A in $\mathbf{C}$.
- $\mathbf{C}_T$ contains an arrow $A_T \xrightarrow{f^\flat} B_T$ for every arrow $A \xrightarrow{f} TB$ in $\mathbf{C}$.
- Given $A_T \xrightarrow{f^\flat} B_T$ and $B_T \xrightarrow{g^\flat} C_T$ in $\mathbf{C}_T$, their composition is defined as:

$$g^\flat \circ f^\flat = (g^* \circ f)^\flat \tag{21}$$

  where

$$g^* = \mu_C \circ Tg \tag{22}$$

- Identity arrows in $\mathbf{C}_T$ are given by:

$$1_{A_T} = (\eta_A)^\flat \tag{23}$$

Then, the adjunction $\langle F_T, G_T, \varphi_T \rangle : \mathbf{C} \rightharpoonup \mathbf{C}_T$ is defined as:

$$F_T : \mathbf{C} \to \mathbf{C}_T = \begin{cases} F_T(A) = A_T \\ F_T(f) = (\eta_B \circ f)^\flat \end{cases} \tag{24}$$

$$G_T : \mathbf{C}_T \to \mathbf{C} = \begin{cases} G_T(A_T) = TA \\ G_T(f^\flat) = f^* \end{cases} \tag{25}$$

$$\varphi_T = f^\flat \mapsto f \tag{26}$$

The unit η_T and counit ε_T of this adjunction are:

$$A \xrightarrow{(\eta_T)_A = \eta_A} TA \tag{27}$$

$$(TA)_T \xrightarrow{(\varepsilon_T)_{A_T} = (1_{TA})^\flat} A_T \tag{28}$$

$$\tag{29}$$