Logical Methods in Computer Science
Volume 15, Issue 2, 2019, pp. 8:1–8:15
https://lmcs.episciences.org/
Submitted Jan. 27, 2018
Published Apr. 30, 2019

REASONING ABOUT EFFECTS: FROM LISTS TO CYBER-PHYSICAL AGENTS

IAN A. MASON AND CAROLYN TALCOTT

SRI International,Menlo Park, CA 94025, USA
e-mail address: {ian.mason,carolyn.talcott}@sri.com

Abstract. Theories for reasoning about programs with effects initially focused on basic manipulation of lists and other mutable data. The next challenge was to consider higher-order programming, adding functions as first class objects to mutable data. Reasoning about actors added the challenge of dealing with distributed open systems of entities interacting asynchronously. The advent of cyber-physical agents introduces the need to consider uncertainty, faults, physical as well as logical effects. In addition cyber-physical agents have sensors and actuators giving rise to a much richer class of effects with broader scope: think of self-driving cars, autonomous drones, or smart medical devices.

This paper gives a retrospective on reasoning about effects highlighting key principles and techniques and closing with challenges for future work.

1. Introduction

"Real programs have effects–creating new structures, examining and modifying existing structures, altering flow of control, etc." This was the first sentence in our 1991 paper published in the debut of the Journal of Functional Programming [21]. According to the Oxford dictionary an effect is "a change that is a result or consequence of an action or other cause." In the computational world, effects can be broadly characterized as Read, Write, or Allocation/Creation effects. Examples include mutable data, objects with local state and methods for access, and actors. The effect, in terms of semantic foundations and reasoning principles, of allowing effects depends on what other capabilities a language or computational model provides, for example: first-order vs higher-order, sequential vs concurrent/distributed.

Fast forward to the present, and the sentence has a much broader meaning in which effects include interacting with and acting on the external environment: self-driving cars and aircraft, medical devices, automated manufacturing, automated biology experiments, smart homes, ...

Equivalence between data structures or active entities is a key concept to be addressed in any system for reasoning about programs. The good news is that the rule "replacing equals by equals gives equals" is usually achievable for a suitable notion of equivalence. The other property of equality that is important in many logics is that replacing a variable by

Key words and phrases: effects, actors, operational equivalence.

DOI:10.23638/LMCS-15(2:8)2019

some expression preserve equality. This fails when evaluation of expressions has effects. What about other laws of equivalence in the presence of effects? What other properties do we want to reason about in general (as opposed to application specific properties)? What about types vs sets defined by a property? What are some helpful reasoning principles or proof schemes?

The work presented here builds on three main themes, corresponding to works that have guided our approach. First, the languages we consider share key features of what we call Landinesque languages in the spirit of Landin's seminal papers [13, 14]. Such languages have a functional core extended by primitives for data and control operations and coupled with an operational semantics structured to support modular extension and equational reasoning. Second, satisfying laws of the computational lambda calculus [27, 28] by the functional core is a key requirements for notions of equivalence. Third, our approach to reasoning and logical formalization is based on Feferman's methods for formalization of constructive mathematics and his ideas concerning variable types [5, 6]. Operational equivalence, being indistinguishable by any enclosing program, has generally been an important approach to defining equivalence of expressions, starting with Plotkin's work [30]. The notion of uniform semantics provides an important tool for deriving laws of operational equivalence, avoiding the need to explicitly reason about "all enclosing programs". In particular, a uniform semantics allows one to compute symbolically with contexts and delay instantiation of variables until they are used.

In the §2 we look at the simplest example of effects: mutable data in a first order language. This was also historically where our study of effects began. We then, in §3, move to the richer world of higher order programming in the presence of mutable data. In both these cases the world is sequential and deterministic. In §4 we look at distributed systems, where the notions of sequentiality, determinism, and even termination no longer play center stage. In §5 we touch on the newer cyber-physical world, and the issues that arise therein. Finally in §6 we summarize and make some concluding remarks on the challenges we have uncovered.

2. First Order Theory of Mutable Data

The simplest examples of effects are those usually slandered as side effects: variable assignment and mutable data such as pointers, arrays, and lisp style lists (or cons cells). Initially we concentrated on the Lisp cons cells, but eventually became more enamored with the ML style reference. `mk` is a memory allocation primitive: the evaluation of $\mathtt{mk}(v)$ results in the allocation of a **new** memory cell and initializes this cell so that it contains the value v. The value returned by this call to `mk` is the newly allocated cell. `mk` is total. `get` is the memory access primitive: the evaluation of $\mathtt{get}(v)$ is defined iff v is a memory cell. If v is a memory cell, then $\mathtt{get}(v)$ returns the value stored in that cell. Note that there is no reason why a cell cannot store itself (or some more elaborate cycle). `get` is partial. `set` is the memory modification primitive: the evaluation of $\mathtt{set}(v_0, v_1)$ is defined iff v_0 is a memory cell. If v_0 is a memory cell, then $\mathtt{set}(v_0, v_1)$ modifies that cell so that its new contents becomes v_1. The value returned by a call to `set` is somewhat arbitrary and somewhat irrelevant. We have chosen `nil` as the return value, thus if v is a cell, then $\mathtt{set}(v, v)$ will return `nil`, and more importantly modify v so that it contains itself. `set` is partial.

Mutable data structures are richer than immutable ones since the ability to mutate allows one to distinguish between objects that have identical structure, but are not the same object stored in memory. The notion of being the same object in memory, in the Lisp tradition, is known as eq-ness, or being eq rather than just equal.

We illustrate this phenomena by providing a function that returns t if the reference cells x and y are the same object in memory, and nil otherwise.

$$\begin{aligned}&\lambda x.\lambda y.\mathtt{let}\{x_0 := \mathtt{get}(x), y_0 := \mathtt{get}(y)\}\\ &\qquad\qquad \mathtt{seq}(\mathtt{set}(x, \mathtt{nil}),\\ &\qquad\qquad\quad \mathtt{set}(y, \mathtt{t}),\\ &\qquad\qquad\quad \mathtt{let}\{z := \mathtt{get}(x)\}\mathtt{seq}(\mathtt{set}(x, x_0), \mathtt{set}(y, y_0), z))\end{aligned}$$

It is important to notice that the function above leaves the state of memory completely unchanged, even though during execution, observable modifications are made. As a result the function would be indistinguishable from the pure version which relies on the eq primitive found in Lisp languages

$$\lambda x.\lambda y.\mathtt{eq}(x, y)$$

assuming a simple (single threaded) notion of indistinguishablity. We can make this observation more formal by using the notion of a context, an expression with a hole, $\bullet$, or more pragmatically an enclosing program. We say two expressions, f and g, are operationally equivalent iff $C[f]$ gives the same result computationally as $C[g]$ for any closing context C in the language at hand. The notion of sameness can usually be taken to be a very coarse grained notion such as simply being defined.

We introduce contexts at this early stage because they turn out to be crucial in the study of languages with effects. They can be used to define the semantics of programs by elegant reduction systems. As we have already seen they can be used to define the notion of computational indistinguishablity, and they can even be used as a logical construct to express properties of programs, akin to a Hoare triple. A contextual assertion takes the form, U$[\![\Phi]\!]$, and asserts that the assertion Φ holds at the point in the computation U when the $\bullet$ is reached. A simple example of this is the axiom which expresses the allocation effects of mk:

$$\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\neg(x \cong y) \wedge \mathtt{cell?}(x) \cong \mathtt{t} \wedge \mathtt{get}(x) \cong v]\!]$$

Intuitively it asserts that the result of a call to $\mathtt{mk}(v)$ is a cell whose contents is v and more importantly, different from every value that existed prior to the call. Contextual assertions are first class formulas and can be quantifed, and be passed to the boolean connectives. Thus we can make the implicitly universally quantified value y explicitly quantified:

$$(\forall y)(\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\neg(x \cong y) \wedge \mathtt{cell?}(x) \cong \mathtt{t} \wedge \mathtt{get}(x) \cong v]\!])$$

This also is a good illustration of the fact that we make no distinction between logical variables, and the variables of our programming language. They are one and the same. We will discuss contextual assertions in more detail in section 3.

In modeling first order languages with mutable data one must have some representation of the current state of the the data structures at hand. In first order Lisp like languages the state of memory can be simply represented by a memory context, an expression, or context, of the form

$$\mathtt{let}\{z_1 := \mathtt{mk}(\mathtt{nil})\} \ldots \mathtt{let}\{z_n := \mathtt{mk}(\mathtt{nil})\}\mathtt{seq}(\mathtt{set}(z_1, v_1), \ldots, \mathtt{set}(z_n, v_n), \bullet)$$

The set of memory contexts, $\mathbb{M}$, is the set of contexts Γ of the above form where $z_i \neq z_j$ when $i \neq j$. Subsequently Γ ranges over $\mathbb{M}$. Here we have used unary cells, the definition for binary cells is entirely analogous. Note that we split the construction of memory into allocation followed by assignment to allow for the construction of arbitrary, possibly cyclic, memory. That memory can be represented as syntactic contexts simplifies the expression of many properties since it provides natural notions of parameterized memory objects, of binding, and of substitution for parameters. We define a reduction calculus on syntactic entities,

$$\Gamma_0; e_0 \mapsto \Gamma_1; e_1$$

called descriptions. They consist of a memory context, a syntactic representation of the state of memory, and an expression, representing the computation taking place. The current computation can be further divided into the current instruction, and the current continuation. Their syntactic counterparts are redexes, and reduction contexts, respectively. Redexes describe the primitive computation steps. A primitive step is either a β_v-reduction or the application of a primitive operation to a sequence of value expressions. Reduction contexts, $\mathbb{R}$, identify the subexpression of an expression that is to be evaluated next, they correspond to the standard reduction strategy (left-first, call-by-value) of [30] and were first introduced in [8]. We use R to range over $\mathbb{R}$.

In addition, the syntactic representation of computation state allows us to compute with open expressions and provides a natural scoping mechanism for memory simply using laws for bound variables. Many of the basic equivalence relations on memories and other semantic entities translate naturally into simple syntactic equivalences such as alpha equivalence.

Reasoning about programs with effects is more delicate than the pure or effect-free languages. For example, it is not the case that substitution instances of equivalent expressions are equivalent $\mathtt{eq}(x, x)$ will always evaluate to $\mathtt{t}$ in a world of atomic data, references, and cons cells, but the substitution instance $\mathtt{eq}(\mathtt{mk}(x), \mathtt{mk}(x))$ will always be false. This is simply because the evaluation of an expression can have effects, and evaluating an expression more than once can be noticeable. Again, one is rescued by contexts, since the property that remains true can be captured by

$$\mathtt{let}\{x := e\}\mathtt{eq}(x, x)$$

always evaluating to $\mathtt{t}$, hinting at the crucial role contexts can make in being able to express subtle properties of the primitives involved.

The quintessential property of operational equivalence is that it is a congruence relation. $e_0 \cong e_1$ implies $C[e_0] \cong C[e_1]$ for any context C, making it an ideal tool for reasoning symbolically about programs with effects. The down side to operational equivalence is that it is in general very hard to establish equivalences. In the case of first order lisp programs this difficulty is surmounted by defining a seemingly stronger perspicuous relation called strong isomorphism, and establishing that it implies operational equivalence.

In [17, 18] strong isomorphism is defined between two expressions e_0 and e_1, written $e_0 \simeq e_1$, if and only if for every closed instantiation[1] the expressions evaluate to equal values in states that are identical, modulo the production of garbage. Here garbage is used to describe memory that is not reachable from either the result, or the original memory. Simple examples of strongly isomorphic expressions are

$$\mathtt{eq}(x, x) \simeq \mathtt{t}$$

$$\mathtt{seq}(\mathtt{set}(x, v), \mathtt{set}(x, w)) \simeq \mathtt{set}(x, w)$$

[1] a closed instantiation is a substitution of values for the free variables that results in a closed expression, where the notion of closed maybe relative to the memory context at hand.

$\mathtt{seq}(\mathtt{mk}(x), \mathtt{mk}(u)) \simeq \mathtt{mk}(u)$

the first two simply evaluate to identical states, the third does so too, but produces some garbage along the way. The main result concerning strong isomorphism, apart from its usefulness in establishing equivalences, is that in the first order Lisp world it coincides with operational equivalence, and so can be used as a tool to establish the operational equivalence of expressions.

In [20, 22] we used this characterization and the ability to reason syntactically to provide a formal system for establishing operational equivalence of first order Lisp like programs, and showed that it was sound. The system was also shown to be complete when restricted to non-recursive programs.

Note that sequentiality is very important in establishing the above results. In a multi-threaded world strong isomorphism would not coincide with operational equivalence, since multi-threaded contexts would be sensitive not just to the result of the computation, but also to the state of the world at every step. Without some form of mutual exclusion one would not be able to define `eq`-ness in terms of mutation, since a process running concurrently could also be mutating the cells being tested.

3. Reasoning about Functions and Effects

Treating functions as first class entities, with the ability to create functions during execution, and to store or return functions as values adds new complications for reasoning about programs. To begin with, equality of values in the usual sense is no longer decidable. This of course is an issue even in the absence of mutable data. Another feature is the ability to create functions that share one or more instances of mutable data structures For example

$\mathtt{let}\{x := \mathtt{mk}(\mathtt{nil})\}\mathtt{let}\{f := \lambda y.e_0(x)\}\mathtt{let}\{g := \lambda z.e_1(x)\}\mathtt{pair}(f, g)$

constructs two closures, f and g, that share a reference cell x. Thus f and g can communicate with each other via setting the value stored in x or export x for other expressions to manipulate. Properties of f and g may crucially depend on how the visibility of this x is maintained, making reasoning a challenge [23, 25]. A concrete example of this is a version of the call-by-value fixed point combinator, let $\mathtt{Y}_v$ be

$$\begin{aligned}&\lambda y.\mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}\\ &\qquad\quad \mathtt{seq}(\mathtt{set}(z, \lambda x.\mathtt{app}(\mathtt{app}(y, \mathtt{get}(z)), x)),\\ &\qquad\qquad \mathtt{get}(z))\end{aligned}$$

This version of the fixed-point combinator is essentially identical to the one suggested by Landin [13]. When applied to a functional F of the form $\lambda f.\lambda x.e$, $\mathtt{Y}_v$ creates a private local cell, z, with contents $G = \lambda x.\mathtt{app}(\mathtt{app}(F, \mathtt{get}(z)), x)$, and returns G. By privacy of z, G is operationally equivalent to $F(G)$ (cf. [21]). Note that this example is typable in the simply typed lambda calculus (for provably non-empty types (cf. [12])). Thus adding operations for manipulating references to the simply typed lambda calculus causes the failure of strong normalization as well as many other of its nice mathematical properties.

As another example, the usual notion of function satisfies the property that each time it is applied to a given argument, the result is the same. This is not the case when functions have memory! Here is a function that returns a different number each time it is called.

$\mathtt{let}\{x := \mathtt{mk}(0)\}\lambda y.\mathtt{let}\{z := \mathtt{get}(x)\}\mathtt{seq}(\mathtt{set}(x, z + 1), z)$

Though simple, such an example can easily be elaborated, using the sieve of Eratosthenes, to enumerate the prime numbers.

3.1. **Equivalence.** Early work on reasoning about equality in higher-order languages includes Plotkin's work defining operational approximations and equivalence for various lambda calculi [30], Felleisen's (and students) work on reduction calculi for languages with effects [7, 9], and Moggi's work on computational monads for a variety of computational primitives [29]. Moggi's equational laws of computational lambda calculus [28] are the core equational theory for lambda-based computational languages.

In [21] we developed a theory of operational approximation and equivalence for a language that combines (call-by-value) lambda calculus and Lisp-like mutable lists. Our definition of operational equivalence extends the extensional equivalence relations defined by Morris and Plotkin to computation over memory structures. Equational laws and methods for proving equivalence were developed building on [16, 22]. This work provided the foundation for a Variable Type Logic of Effects [12] which extended equational reasoning with language for defining sets (properties) and principles for reasoning about set membership.

Just as in § 2, the basis of our definition of operational equivalence is a small step operational semantics, defined using memory contexts to represent memory state and reduction contexts to represent the continuation of a computation and reduction rules to define the small steps of a computation. Two expressions e_0, e_1 are operationally equivalent if for any closing context C, $C[e_0]$ and $C[e_1]$ are equi-defined. This looks identical to the definition in the first-order case. The difference is in the set of possible contexts. It is easy to see that this is a congruence relation so substitution of equals for equals gives equals. But, substitution of an expression into equals does not give equals. The counter-example shown in § 2 remains a counter-example.

The definition of strong isomorphism in the first order case can be lifted to our higher-language in an entirely analogous fashion and just as in the first-order case, we have that

- strong isomorphism implies operational equivalence.

A key feature of $\simeq$ is that reduction rules of the operational semantics are a subset of the $\simeq$ relation. Thus many laws can be proved by showing two expressions have a common reduct. For example

$$\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{seq}(\mathtt{set}(z, w), e) \simeq \mathtt{let}\{z := \mathtt{mk}(w)\}e$$

if z and w are distinct variables. Furthermore, many of the laws of strong isomorphism from the first-order case continue to hold as laws of operational equivalence in the higher-order case, including laws based on reductions that do not directly involve functions.

The $\simeq$ laws combined with congruence entail that the η law of the lambda calculus holds in the sense that if e denotes a function, i.e. $e \cong \lambda x.e_0$, then $e \cong \lambda x.e(x)$. In contrast, if we view the notion of function in the more general sense of being a lambda with local memory, $e \cong \Gamma[\lambda x.e_0]$, then the η law fails. That is, in general $\lambda x.(\Gamma[\lambda x.e])x$ is not operationally equivalent to $\Gamma[\lambda x.e]$. As a counter-example take Γ to be $\mathtt{let}\{z := \mathtt{mk}(0)\}\bullet$ and $\lambda x.e$ to be $\lambda x.\mathtt{let}\{y := \mathtt{get}(z)\}\mathtt{seq}(\mathtt{set}(z, x), y)$. Since in this case $\lambda x.(\Gamma[\lambda x.e])x$ is operationally equivalent to $\lambda x.0$, albeit with a memory leak, while $\Gamma[\lambda x.e]$ is a thunk that when applied returns the value it was previously applied to.

So how does the presence of higher-order entities distinguish $\cong$ vs $\simeq$? Take any two distinct operationally equivalent lambda expressions, the simplest pair that comes to mind is:

$\lambda x.x$ and $\lambda x.\mathtt{seq}(\mathtt{mk}(0), x)$, these are operationally equivalent, but not strongly isomorphic because, as values, to be strongly isomorphic, they would have to be identical.

Strong isomorphism and computational reasoning based on reduction rules nicely capture laws of local data and memory manipulation but there is much more to operational equivalence and reasoning about all contexts is daunting, even in the absence of memory structures. Robin Milner's context lemma [26] showed that operational equivalence can be proved by considering a small number of context patterns, thus greatly reducing the complexity of proving operational equivalence laws.

An analog to the context lemma for languages with effects is the CIU (Closed Instantiations of Uses) theorem which states that

- if all closed instantiations of all uses of two expressions are equidefined then the expressions are operationally equivalent.

A closed instantiation of a use of an expression e is a closed expression of the form $\Gamma[R[e^\sigma]]$ where the memory context Γ and substitution σ represent the closed instantiation and the reduction context R represents the use. As hinted in the introduction, uniform semantics is key to proving CIU. Once established, CIU is used to develop methods for proving equivalence of lambda functions with and without memory.

Using this theorem we can easily establish, for example, the validity of the Moggi's let-rules of the computational lambda calculus [28] (see also [33] where these laws are established for a language with control abstractions).

(i) $\mathtt{app}(\lambda x.e, \mathrm{v}) \cong e^{\{x:=\mathrm{v}\}} \cong \mathtt{let}\{x := \mathrm{v}\}e$

(ii) $\mathrm{R}[e] \cong \mathtt{let}\{x := e\}\mathrm{R}[x]$

(iii) $\mathrm{R}[\mathtt{let}\{x := e_0\}e_1] \cong \mathtt{let}\{x := e_0\}\mathrm{R}[e_1]$

where in (ii) and (iii) we require x not free in R.

Another nice property that is easily established using CIU is that reduction preserves operational equivalence:

$$\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; \Gamma[e] \cong \Gamma'[e']$$

This property is the basis of the calculi found in [10]. Our lambda language is an example of a Landinesque language, so called in the spirit of Landin's "Next 700 Programming languages" paper [14]. A key result is that the CIU theorem holds for any Landinesque language with a suitably nice (uniform) semantics. Uniformity is captured by the ability to compute with contexts rather than just expressions, and the exact notion of uniformity is axiomatized in [24].

3.2. **Formulas.** In addition to being a useful tool for establishing laws of operational equivalence, CIU can be used to define a satisfaction relation between memory contexts and equivalence assertions. In an obvious analogy with the usual first-order Tarskian definition of satisfaction this can be extended to define a satisfaction relation $\Gamma \models \Phi[\sigma]$ for formulas Φ and closing substitutions σ [2].

The memory context Γ plays the role of the model, in that it specifies what objects exist in memory, while the closing substitution σ binds variables to values that exist in that model. Note that variables bound by the memory context Γ are cells, while variables bound

[2] Here a closing substitution binds at least the free variables not bound in the memory context.

by the substitution σ are arbitrary values. The adjective closing just emphasizes that all free variables of Γ and Φ are in the domain of the substitution, and that no free variables creep in amongst the values in the range of σ.

The atomic formulas of our language assert the operational equivalence of two expressions. In addition to the usual first-order formula constructions we add contextual assertions: if Φ is a formula and U is a certain type of context, then $\mathrm{U}[\![\Phi]\!]$ is a formula. This form of formula expresses the fact that the assertion Φ holds at the point in the program text marked by the hole in U, if execution of the program reaches that point. The contexts allowed in contextual assertions are called univalent contexts, (U-contexts). They are the largest natural class of contexts whose symbolic evaluation is unproblematic. The key restriction is that we forbid the hole to appear in the scope of a (non-`let`) lambda, thus preventing the proliferation of holes.

One simple consequence of the definitions are the following three principles for reasoning about contextual assertions: a general principle for introducing contextual assertions (akin to the rule of necessitation in modal logic); a principle for propagating contextual assertions through equations; and a principle for composing contexts (or collapsing nested contextual assertions).

(i) $\models \Phi \quad \text{implies} \quad \models \mathrm{U}[\![\Phi]\!]$

(ii) $\mathrm{U}[\![e_0 \cong e_1]\!] \;\Rightarrow\; \mathrm{U}[e_0] \cong \mathrm{U}[e_1]$

(iii) $\mathrm{U}_0[\![\mathrm{U}_1[\![\Phi]\!]]\!] \;\Leftrightarrow\; (\mathrm{U}_0[\mathrm{U}_1])[\![\Phi]\!]$

Also, as we have already mentioned in section 2 one can naturally express properties such as the allocation effects of `mk`:

$$(\forall y)(\mathtt{let}\{x := \mathtt{mk}(\mathrm{v})\}[\![\neg(x \cong y) \;\wedge\; \mathtt{cell?}(x) \cong \mathtt{t} \;\wedge\; \mathtt{get}(x) \cong \mathrm{v}]\!])$$

3.3. **Classes.** Using methods developed by Feferman [5,6] and applied to lambda languages with control operators [33], we extend our theory to include a general theory of classifications (classes for short). With the introduction of classes, principles such as structural induction, as well as principles accounting for the effects of an expression can easily be expressed. Classes serve as a starting point for studying semantic notions of type. As will be seen, direct representation of type inference systems can be problematic, and additional notions maybe required to provide a formal semantics. Even here classes are likely to play an important role.

Class terms are either class variables, class constants, or comprehension terms, $\{x \mid \Phi\}$. We extend the set of formulas to include class membership and quantification over class variables. We define (extensional) equality and subset relations on classes in the usual manner.

$$K_0 \subseteq K_1 \quad \text{abbreviates} \quad (\forall x)(x \in K_0 \;\Rightarrow\; x \in K_1)$$

$$K_0 \equiv K_1 \quad \text{abbreviates} \quad K_0 \subseteq K_1 \;\wedge\; K_1 \subseteq K_0$$

A simple example of a class is the set of reference cells that contain values in a specific set K:

$$\mathbf{Cell} = \{x \mid \mathtt{cell?}(x) \cong \mathtt{t}\}$$

$$\mathbf{Cell}[K] = \{x \mid \mathtt{cell?}(x) \cong \mathtt{t} \;\wedge\; \mathtt{get}(x) \in K\}$$

We can also express a variety of function spaces, the simplest are total, partial and memory.[3]

$$\bar{X} \to Y = \{f \mid (\forall \bar{x} \in \bar{X})(\exists y \in Y)\mathtt{app}(f, \bar{x}) \cong y\}$$

$$\bar{X} \xrightarrow{\mathrm{p}} Y = \{f \mid (\forall \bar{x} \in \bar{X})(\forall y)(\mathtt{app}(f, \bar{x}) \cong y \Rightarrow y \in Y)\}$$

$$\bar{X} \xrightarrow{\mu} Y = \{f \mid (\forall \bar{x} \in \bar{X})(\mathtt{let}\{y := \mathtt{app}(f, \bar{x})\}[\![y \in Y]\!])\}$$

So for example, the reference operations can be given types by

(mk) $\lambda x.\mathtt{mk}(x) \in (X \xrightarrow{\mu} \mathbf{Cell}[X])$

(get) $\lambda x.\mathtt{get}(x) \in \mathbf{Cell}[X] \to X$

(set) $\lambda x.\lambda y.\mathtt{set}(x, y) \in \mathbf{Cell} \to (\mathbf{Val} \xrightarrow{\mu} \mathbf{Nil})$

Class membership expresses a very restricted form of non-expansiveness, allowing neither expansion of memory domain nor change in contents of existing cells. To illustrate some of the subtleties regarding class membership, and notions of expansiveness, consider the following expressions:

$$e_0 = \lambda x.\mathtt{mk}(\mathtt{nil})$$

$$e_1 = \mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}\lambda x.z$$

$$e_2 = \mathtt{seq}(\mathtt{if}(\mathtt{cell?}(y), \mathtt{set}(y, \mathtt{nil}), \mathtt{nil}), \lambda x.\mathtt{mk}(\mathtt{nil}))$$

$$e_3 = \mathtt{seq}(\mathtt{if}(\mathtt{cell?}(y), \mathtt{set}(y, \mathtt{nil}), \mathtt{nil}), \mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}\lambda x.z)$$

Then each of these expressions evaluates to a memory function mapping arbitrary values to cells containing `nil`. But they differ in the effects they have. e_0 is a value (and as such neither expands nor modifies memory). e_1 is not a value and is expansive (its evaluation enlarges the domain of memory) but does not modify existing memory. e_2 may modify existing memory, but does not expand it. e_3 is expansive, and it may modify existing memory. These observations can be expressed in the theory as follows. Let T be $\mathbf{Val} \xrightarrow{\mu} \mathbf{Cell}[\mathbf{Nil}]$, and $\Phi_{\neg\mathrm{write}}[\mathbf{Cell}](e)$ be as defined below. Then

$$e_0 \in T \wedge e_0 \in \mathbf{Val}$$

$$e_j \notin \mathbf{Val} \quad \text{for} \quad 1 \leq j \leq 3$$

$$\mathtt{let}\{x := e_j\}[\![x \in T]\!] \quad \text{for} \quad 0 \leq j \leq 3$$

$$\Phi_{\neg\mathrm{write}}(e_j) \quad \text{for} \quad 0 \leq j \leq 1$$

$$\Phi_{\neg\mathrm{expand}}(e_j) \quad \text{for} \quad j \in \{0, 2\}$$

Let $\Phi_{\neg\mathrm{expand}}(e)$ stand for the formula

$$(\forall X)(X \equiv \mathbf{Cell} \Rightarrow \mathtt{seq}(e, [\![X \equiv \mathbf{Cell}]\!]).$$

Then $\Phi_{\neg\mathrm{expand}}(e)$ says that execution of e does non expand the memory, although it might modify contents of existing cells. $\Phi_{\neg\mathrm{write}}(e)$ is defined as:

$$(\exists X)(X \equiv \mathbf{Cell} \wedge \\ (\forall x \in X)(\forall z \in \mathbf{Val}))(\mathtt{get}(x) \cong z \Rightarrow \mathtt{seq}(e, [\![\mathtt{get}(x) \cong z]\!])))$$

[3]We use the standard notation of $\bar{x}$ to denote a sequence $x_0, \ldots, x_n$ of variables.

3.4. **Classes vs Types: the functional case.** In [6] Feferman proposes an explanation of ML types in the variable type framework. This gives a natural semantics to ML type expressions, but there are problems with polymorphism, even in the purely functional case. The collection of classes is much too rich to be considered a type system. One problem that arises is that fixed-point combinators can not be uniformly typed over all classes. This problem arises even in the absence of memory [31,33]. Let $\mathtt{Y}_v$ by any fixed-point combinator (such that $f(\mathtt{Y}_v(f)) = \mathtt{Y}_v(f)$). Then it is not the case that

$$f \in (C \to C) \;\Rightarrow\; \mathtt{Y}_v(f) \in C$$

for all function classes C ($C \subseteq A \stackrel{\text{p}}{\to} B$ for some classes A, B).

Define P to be the class of strictly partial maps from **Nat** to **Nat**:

$$P = \{g \in \mathbf{Nat} \stackrel{\text{p}}{\to} \mathbf{Nat} \mid (\exists n \in \mathbf{Nat})(\neg \downarrow g(n))\}$$

Let

$$f = \lambda p.\lambda n.\mathtt{if}(\mathtt{eq}(n,0), n, p(n-1))$$

Then we can prove

(1) $\quad f \in P \to P$

(2) $\quad \mathtt{Y}_v(f) \in \mathbf{Nat} \to \mathbf{Nat}$

(**1**) follows by simple properties of `if`, `eq` and arithmetic (**2**) follows by induction on **Nat** using the fixed point property of Y. Consequently, $\neg(\mathtt{Y}_v(f) \in P)$

3.5. **Classes vs Types: the imperative case.** The situation becomes more problematic when references are added, even in the simply typed (or monomorphic) case. Naïve attempts to represent ML types as classes fails in sense that ML inference rules are not valid. The essential feature of the ML type system, in addition to the inference rules, is the preservation of types during the execution of well-typed programs, not just of the text being executed, but also of the contents of any cell in memory. This requirement is a strong form of subject reduction. One that does not seem to be expressible using classes (quantifying over types, whatever they may be, seems problematic). Our analysis indicates that ML types are therefore more syntactic than semantic.

4. Actors: Open Systems of Interactive Agents

An actor is a unit of concurrent/distributed interactive computation. Each actor encapsulates state. It can receive messages; which may cause it to change state; it can send messages, to actors it knows about; and it can create new actors. Communication by message passing is reliable, and asynchronous with fair message delivery [1, 11]. We can describe actor behaviors using lambda expressions augmented with actor primitives (`become`, `send` and `letactor`) analogous to describing computation over memory structures by adding memory effect primitives [2]. `send` is for sending messages; $\mathtt{send}(a, v)$ creates a new message with receiver a and contents v and puts the message into the message delivery system. `letactor` is for actor creation. $\mathtt{letactor}\{x := b\}e$ creates an actor with initial behavior b, making the new address the value of the variable x. The expression e is evaluated in the extended environment. The variable x is also bound in the expression b, thus allowing an actor to refer to itself if so desired. `become` is for changing behavior; $\mathtt{become}(b)$ creates an anonymous actor to carry out the rest of the current computation, alters the behavior of the actor executing the become to be b, and frees that actor to accept another message. This provides additional parallelism. The anonymous actor may send messages or create new actors in the process of completing its computation, but will never receive any messages as its address can never be known.

A consequence of the actor interaction model is unbounded non-determinism. A classic example is the Ticker actor that maintains a counter, sends itself tick messages to increment the counter, and responds to requests from other actors by sending the current counter value.

$$
\begin{aligned}
b_{\text{Ticker}} = \mathtt{Y}_v(\lambda b.\lambda c.\lambda m.\mathtt{if}(m = \mathit{tick},\\
&\mathtt{seq}(\mathtt{send}(\tau, \mathit{tick}), \mathtt{become}(\mathtt{app}(b, c+1)))\\
&\mathtt{seq}(\mathtt{send}(\mathit{customer}(m), c), \mathtt{become}(\mathtt{app}(b, c)))))
\end{aligned}
$$

$$
\text{Ticker} = \mathtt{letactor}\{\tau := b_{\text{Ticker}}\}\mathtt{seq}(\mathtt{send}(\tau, \mathit{tick}), \tau)
$$

We avoid going into the details of messages as data structures by using $\mathit{customer}(m)$ to denote the sender of the message. The Ticker has the property that (assuming it is sent an initial tick message) for any natural number n there is a computation where a request results in sending a number greater than n. This is because, although the request is guaranteed to be delivered and receive a response, any number of ticks can be delivered before the request.

The operational semantics for actor systems is given by a transition relation on actor configurations. A configuration

$$\left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle_{\xi}^{\rho}$$

can be thought of as representing a global snapshot of an actor system with respect to some idealized observer [1]. It contains a collection of actors α, messages μ, external actor names ξ , and receptionist names ρ. The sets of receptionists and external actors are the interface of an actor configuration to its environment. They specify which actors are visible and which actor connections must be provided for the configuration to function. Both the set of receptionists and the set of external actors may grow as the configuration evolves.

Several semantics have been defined for actor configurations [35] differing by treatment of ordering relations among send/receive events and level of detail [3]. The basic operational semantics is the set of traces of fair executions given by a reduction relation as for Landinesque languages. What is different is the presence of interactions with the external world – transitions for input of messages from external (unseen) actors (in(msg)), and output of messages to these external actors (out(msg)).

Although we have never done so, actor computation is uniform enough for it to be represented as a Landinesque language, what is lacking is the development of a syntax rich enough to represent configurations. However, a somewhat more crucial distinction is that unlike the sequential case, neither the notion of reducing to a value, nor deterministic computation, nor the notion of a computation terminating are central concepts. Rather they are side lined to the more infinitary notion of a computation path, and the collection of all computation paths. It is in this infinitary realm that crucial questions of fairness arise and play a part. The unimportance of termination creates a new problem: what are the primitive observations that underly any notion of operational or observational equivalence? The approach taken in [2] is to introduce a primitive, event, and observe whether or not in a given computation, event is executed. This approach is similar in spirit to that used in defining testing equivalence for CCS [4], except that the required condition of fairness of actor computation simplifies matters by collapsing two obvious candidates of equivalence into one. See §4 of [2] for more details.

With a notion of equivalence on actor expressions defined, a library of useful equivalences can be established. Since our reduction rules preserve the evaluation semantics of the embedded functional language, many of the equational laws for the language of section 3 (cf. [34]) continue to hold in the actor language. For example, the laws of the untyped computational lambda calculus [27] continue to hold in the actor setting [2].

Even though the actor language is not presented as a Landinesque language, the fact that computation can be parametrically defined more generally on contexts allows for laws to be established in an entirely analogous fashion to the CIU principle. For example if there is some e' such that $\mathrm{R}_0[x] \overset{*}{\mapsto} e'$ and $\mathrm{R}_1[x] \overset{*}{\mapsto} e'$ where x is a fresh variable, then $\mathrm{R}_0[e] \cong \mathrm{R}_1[e]$ for any e. This rule says that if two reduction contexts have a common λ-reduct when the redex hole is filled with a fresh variable (standing for an arbitrary value expression), then they are equivalent. In other words, two reduction contexts are considered equivalent if placing an arbitrary expression in the redex hole results in equivalent expressions.

5. Cyber-Physical Agents

Actors are an idealization of real world interactive agents: messages are always delivered, intact, to the right actor. The interactions are simply exchanges of information. Autonomous cyber-physical agents (CPAs) combine interaction as information exchange with interaction with the physical world via sensors and actuators. Examples include drones used for agricultural surveillance, railway track monitoring, or package delivery; security robots, wave gliders that traverse the Pacific Ocean by themselves; and self-driving cars. CPAs interact in space and time and have finite resources. Things don't always work as expected: sensors may give false readings; actuators (driving engines, rotors, cameras) may fail to act or cause too much or too little effect; or there may be natural threats such as obstacles or bad weather impeding a mobile CPA. Communication is likely to be disrupted so coordination amongst agents is a challenge.

In the actor model the notion of fairness attempts to capture that actors are independent agents running on independent clocks combined with reliable message delivery. It ensures that one actor does not get all the resources in a situation of concurrent processing on a shared host. In the case of autonomous CPAs we are modeling physically independent agents. Fairness is in some sense built in to the physics. Although agents can purposely interfere with one another, that is a behavior problem, not a model problem. Also, fairness is an infinitary property, and limits of the sort used to define fairness aren't observable in the real world. From a practical point of view, we are typically interested in behaviors of CPAs over a finite time horizon, in which case fairness, being an infinitary property, does not play a role.

To define an interaction path semantics for CPA systems, one needs semantic rules for agent behaviors, which include rules modeling the physical effects of sensors and actuators, rules modeling relevant aspects of the external environment. Examples can be found in [15, 19].

To define operational equivalence in analogy to actor systems we would need a notion of closing configuration. It is not clear that there is in general a meaningful such notion. If the rules for sensors and actuators capture fault/threat models they are likely to be probabilistic, leaving the question of what to check about the set of interaction paths to decide equivalence. Is it interesting or useful to have a probability measure on equivalence?

We propose that a first approach to reasoning about CPAs is to identify effects that we are concerned with, and use these to formulate goals that a CPA system should achieve. Examples of goals include monitoring (taking a picture or sampling air or water for quality assessment, checking inventory); moving objects; not running out of energy, not doing damage. Monitoring goals come with space and time requirements. Achievement of goals is not all or nothing, but can be measured either in a discrete or continuous (partially) ordered domain. For example the percent of specified locations visited or sampled by a monitor system in a given period could be a measure of achievement. Another measure could be the percent of energy remaining or the minimal energy reserve at any point in carrying out a task. These could be combined lexicographically giving preference to safety to give an overall measure of success. See [19, 32] for examples. Given such measures, one could compare CPAs based on how well they achieve goals, leading to a partial order on CPAs. In different circumstances the ordering of importance of goals may change and thus the ranking of agents may change.

6. Conclusion

Effects are an essential part of interaction and communication. In computation systems effects are observed by and affect the remaining computation (continuation), the concurrent computations, and observers outside the system.

From studies that develop theories of effects, key concepts for formalizing and reasoning about programs with effects have emerged. These include a variety of contexts (reduction, memory, closing, ...); reduction calculii, and operational notions of equivalence. The ability to represent execution state as contexts leads to an elegant operational semantics, and is also key for further developing the theory of effects. Uniformity – reduction rules that are uniformly parameterized by the surrounding context – is a powerful tool for developing reasoning principles; an example of this is the CIU theorem.

Equivalence and the consequences of effects are very sensitive to the richness of the contexts. Contexts have dimensions beyond what is normally thought of as effects, including: first-order versus higher-order, functions can encapsulate and replicate effects as they are passed around; sequential versus concurrent/distributed, introducing the complications of non-determinism and interference mid-computation.

In each case some equational laws will break. However, the laws of the computational lambda calculus hold in all cases where there is a uniform semantics, an indication of the importance of that calculus as a core for computational languages.

Logics for reasoning about programs/systems with effects have been developed building on the equational theories. Again contexts are key for axiomatizing effectual primitives and for expressing properties such as invariants. There are completeness results for first-order fragments. In other cases a combination of computational and logical reasoning seems useful, taking advantage again of reasoning principles based on uniform computation.

There remain a number of interesting challenges for reasoning about effects. One example is relating syntactic and semantic notions of type. Are there semantic types that can be checked by syntactic type rules? Are there syntactic types that have semantic characterizations.

Syntactic representation of execution contexts has been a crucial tool for developing reasoning methods. Although the formal development has not been done for actor languages, the reasoning methods relied on a mix of syntactic and semantic contexts that make it clear a fully syntactic representation of computation contexts is possible. This remains an open question for cyber-physical agents (CPAs). Perhaps some form of symbolic reasoning where the unknown parts of the context remain unspecified?

Reasoning about CPAs introduces many new issues as a consequence of the physical nature of effects and interacting in an open unpredictable environment. Sensors and actuators may be faulty, other agents and nature may interfere. Furthermore, some cyber/digital effects disappear when system stops (files, databases, hopefully do not). Effects caused by CPAs may persist after the system task ends, by design or due to errors, until another system (CPA, nature, human) causes further change.

References

[1] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Mass., 1986.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. Journal of Functional Programming, 7:1–72, 1997.

[3] H. G. Baker and C. Hewitt. Laws for communicating parallel processes. In IFIP Congress, pages 987–992. IFIP, Aug. 1977.

[4] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. Theoretical Computer Science, 34:83–133, 1984.

[5] S. Feferman. A language and axioms for explicit mathematics. In Algebra and Logic, volume 450 of Springer Lecture Notes in Mathematics, pages 87–139. Springer Verlag, 1975.

[6] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In Logic and Computation, volume 106 of Contemporary Mathematics, pages 101–136. A.M.S., Providence R. I., 1990.

[7] M. Felleisen. The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages. PhD thesis, Indiana University, 1987.

[8] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, Formal Description of Programming Concepts III, pages 193–217. North-Holland, 1986.

[9] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. Theoretical Computer Science, 52:205–237, 1987.

[10] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science, 103:235–271, 1992.

[11] C. Hewitt. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. PhD thesis, MIT, 1971.

[12] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. Information and Computation, 119(1):55–90, 1995.

[13] P. J. Landin. The mechanical evaluation of expressions. Computer Journal, 6:308–320, 1964.

[14] P. J. Landin. The next 700 programming languages. Comm. ACM, 9:157–166, 1966.

[15] M. Loreti and J. Hillston. Modelling and analysis of collective adaptive systems with carma and its tools. In SFM 2016: Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems, volume 9700 of LNCS. Springer, 2016.

[16] I. A. Mason. The Semantics of Destructive Lisp. PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[17] I. A. Mason. The Semantics of Destructive Lisp. Number 5 in CSLI Lecture Notes, Center for the Study of Language and Information, Stanford University. University of Chicago Press, 1986.

[18] I. A. Mason. Verification of Programs that Destructively Manipulate Data. Science of Computer Programming, 10:177–210, 1988.

[19] I. A. Mason, V. Nigam, C. Talcott, and A. Brito. A framework for analyzing adaptive autonomous aerial vehicles. In 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems, 2017.

[20] I. A. Mason and C. L. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Department of Computer Science, Stanford University, 1989.
[21] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. Journal of Functional Programming, 1:287–327, 1991.
[22] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. Theoretical Computer Science, 105(2):167–215, 1992.
[23] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In Seventh Annual Symposium on Logic in Computer Science, pages 186–197. IEEE, 1992.
[24] I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer, and C. Talcott, editors, Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman, Lecture Notes in Logic, pages 299–344. Association of Symbolic Logic, 2002.
[25] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In 15th ACM Symposium on Principles of Programming Languages, pages 191–208, 1988.
[26] R. Milner. Fully abstract models of typed λ-calculi. Theoretical Computer Science, 4:1–22, 1977.
[27] E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.
[28] E. Moggi. Computational lambda-calculus and monads. In Fourth Annual Symposium on Logic in Computer Science. IEEE, 1989.
[29] E. Moggi. Notions of computation and monads. Information and Computation, 93(1), 1991.
[30] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. Theoretical Computer Science, 1:125–159, 1975.
[31] S. F. Smith. Partial Objects in Type Theory. PhD thesis, Cornell University, 1988. Available as TR 88-938.
[32] C. Talcott, F. Arbab, and M. Yadav. Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, volume 8950 of LNCS. Springer-Verlag, 2015.
[33] C. L. Talcott. A theory for program and data type specification. Theoretical Computer Science, 104:129–159, 1992.
[34] C. L. Talcott. A theory for program and data specification. Theoretical Computer Science, 104:129–159, 1993.
[35] C. L. Talcott. Composable semantic models for actor theories. Higher-Order and Symbolic Computation, 11(3):281–343, 1998.