A Sound and Complete Logic for Algebraic Effects

Cristina Matache([✉]) and Sam Staton

University of Oxford, Oxford, UK
cristina.matache@balliol.ox.ac.uk

Abstract. This work investigates three notions of program equivalence for a higher-order functional language with recursion and general algebraic effects, in which programs are written in continuation-passing style. Our main contribution is the following: we define a logic whose formulas express program properties and show that, under certain conditions which we identify, the induced program equivalence coincides with a contextual equivalence. Moreover, we show that this logical equivalence also coincides with an applicative bisimilarity. We exemplify our general results with the nondeterminism, probabilistic choice, global store and I/O effects.

1 Introduction

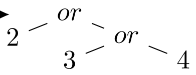
Logic is a fundamental tool for specifying the behaviour of programs. A general approach is to consider that a logical formula ϕ encodes a program property, and the satisfaction relation of the logic, $t \models \phi$, asserts that program t enjoys property ϕ. An example is Hennessy-Milner logic [12] used to model concurrency and nondeterminism. Other program logics include Hoare logic [13], which describes imperative programs with state, and more recently separation logic [28]. Both state and nondeterminism are examples of *computational effects* [25], which represent impure behaviour in a functional programming language. The logics mentioned so far concern languages with first-order functions, so as a natural extension, we are interested in finding a logic which describes higher-order programs with general effects.

The particular flavour of effects we consider is that of *algebraic effects* developed by Plotkin and Power [32–34]. This is a unified framework in which effectful computation is triggered by a set of operations whose behaviour is axiomatized by a set of equations. For example, nondeterminism is given by a binary choice operation $or(-,-)$ that satisfies the equations of a semilattice. Thus, general effectful programs have multiple possible execution paths, which can be visualized as an (effect) tree, with effect operations labelling the nodes. Consider the following function `or_suc` which has three possible return values, and the effect tree of (`or_suc` 2):

M. Bojańczyk and A. Simpson (Eds.): FOSSACS 2019, LNCS 11425, pp. 382–399, 2019.
https://doi.org/10.1007/978-3-030-17127-8_22

$$\texttt{or\_suc} = \lambda x{:}\texttt{nat}.\quad \mathit{or}(x,\ \mathit{or}(x+1, x+2))$$

$(\texttt{or\_suc}\ 2) \longmapsto$



Apart from state and nondeterminism, examples of algebraic effects include probabilistic choice and input and output operations.

Apart from providing a specification language for programs, a logic can also be used to compare two different programs. This leads to a notion of program equivalence: two programs are equivalent when they satisfy exactly the same formulas in the logic.

Many other definitions of program equivalence for higher-order languages exist. An early notion is contextual equivalence [26], which asserts that two programs are equivalent if they have the same observable behaviour in all program contexts. However, this is hard to establish in practice due to the quantification over all contexts. Another approach, which relies on the existence of a suitable denotational model of the language, is checking equality of denotations. Yet another notion, meant to address the shortcomings of the previous two, is that of applicative bisimilarity [1].

Given the wide range of definitions of program equivalence, comparing them is an interesting question. For example, the equivalence induced by Hennessy-Milner logic is known to coincide with bisimilarity for CCS. Thus, we not only aim to find a logic describing general algebraic effects, but also to compare it to existing notions of program equivalence.

Program equivalence for general algebraic effects has been studied by Johann, Simpson and Voigtländer [17] who define a notion of contextual equivalence and a corresponding logical relation. Dal Lago, Gavazzo and Levy [7] provide an abstract treatment of applicative bisimilarity in the presence of algebraic effects. Working in a typed, call-by-value setting, Simpson and Voorneveld [38] propose a modal logic for effectful programs whose induced program equivalence coincides with applicative bisimilarity, but not with contextual equivalence (see counterexample in Sect. 5). Dal Lago, Gavazzo and Tanaka [8] propose a notion of applicative similarity that coincides with contextual equivalence for an untyped, call-by-name effectful calculus.

These papers provide the main starting point for our work. Our goal is to find a logic of program properties which characterizes contextual equivalence for a higher-order language with algebraic effects. We study a typed call-by-value language in which programs are written in continuation-passing style (CPS). CPS is known to simplify contextual equivalence, through the addition of control operators (e.g. [5]), but it also implies that all notions of program equivalence we define can only use continuations to test return values. Contextual equivalence and bisimilarity for lambda-calculi with control, but without general effects, have been studied extensively (e.g. [4,15,23,41]).

In CPS, functions receive as argument the continuation (which is itself a function) to which they pass their return value. Consider the function that adds two natural numbers. This usually has type $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$, but its CPS version is defined as: $\texttt{addk} = \lambda(n{:}\texttt{nat}, m{:}\texttt{nat}, k{:}\texttt{nat}{\to}\texttt{R}).\ k\ (n+m)$ for some fixed return type R. The function or_suc becomes in CPS:

$$\texttt{or\_succ} = \lambda(x{:}\texttt{nat}, k{:}\texttt{nat}{\to}\texttt{R}).\ \mathit{or}(k\ x,\ \mathit{or}(\texttt{addk}\ (x,\ 1,\ k),\ \texttt{addk}\ (x,\ 2,\ k))).$$

A general translation of direct-style functions into CPS can be found in Sect. 5.

We fix a calculus named ECPS (Sect. 2), in which programs are not expected to return, except through a call to the continuation. Contextual equivalence is defined using a custom set of observations $\mathfrak{P}$, where the elements of $\mathfrak{P}$ are sets of effect trees. We consider a logic $\mathcal{F}$ whose formulas express properties of ECPS programs (Sect. 3). For example, `or_succ` satisfies the following formula: $\phi = (\{2\},\ (\{3\} \vee \{4\}) \mapsto \Box) \mapsto \Diamond$.

Here, $\Diamond$ is the set of all effect trees for which at least one execution path succeeds and $\Box$ is the set of trees that always succeed. So `or_succ` $\models_{\mathcal{F}} \phi$ says that, when given arguments 2 and a continuation that always succeeds for input 3 or 4, then `or_succ` *may* succeed. In other words, `or_succ` may 'return' 3 or 4 to the continuation. In contrast, `or_succ` $\models_{\mathcal{F}} \phi' = (\{2\},\ (\{3\} \vee \{4\}) \mapsto \Box) \mapsto \Box$ says that the program `or_succ` *must* return 3 or 4 to the continuation. Thus `or_succ` $\not\models_{\mathcal{F}} \phi'$ because the continuation k might diverge on 2.

Another example can be obtained by generalizing the `or_succ` function to take a function as a parameter, rather than using `addk`:

$$\begin{aligned}\texttt{or\_succ'} = \lambda(x : \texttt{nat},\ k : \texttt{nat}{\to}\texttt{R},\ f : (\texttt{nat},\ \texttt{nat},\ \texttt{nat}{\to}\texttt{R}){\to}\texttt{R}).&\\ \mathit{or}(k\ x,\ \mathit{or}(f\ (x, 1, k),\ f\ (x, 2, k)))&\\ \models_{\mathcal{F}} \Big(\{2\},\ \{4\} \mapsto \Diamond,\ ((\{2\},\ \{2\},\ \{4\} \mapsto \Diamond) \mapsto \Diamond)\Big) \mapsto \Diamond.&\end{aligned}$$

The formula above says that `or_succ'` may call f with arguments 2, 2 and k.

The main theorem concerning the logic $\mathcal{F}$ (Theorem 1) is that, under certain restrictions on the observations in $\mathfrak{P}$, logical equivalence coincides with contextual equivalence. In other words, $\mathcal{F}$ is sound and complete with respect to contextual equivalence. The proof of this theorem, outlined in Sect. 4, involves applicative bisimilarity as an intermediate step. Thus, we show in fact that three notions of program equivalence for ECPS are the same: logical equivalence, contextual equivalence and applicative bisimilarity. Due to space constraints, proofs are omitted but they can be found in [21].

2 Programming Language – ECPS

We consider a simply-typed functional programming language with general recursion, a datatype of natural numbers and general algebraic effects as introduced by Plotkin and Power [32]. We will refer to this language as ECPS because programs are written in continuation-passing style.

ECPS distinguishes between terms which can reduce further, named computations, and values, which cannot reduce. ECPS is a variant of both Plotkin's PCF [31] and Levy's Jump-With-Argument language [20], extended with algebraic effects. A fragment of ECPS is discussed in [18] in connection with logic.

Types	$A, A_1, B \coloneqq (A_1, \ldots, A_n){\to}\texttt{R} \mid \texttt{nat}$	$(n \geq 0)$
Typing contexts	$\Gamma \coloneqq \emptyset \mid \Gamma, x : A.$	

The only base type in ECPS is $\mathtt{nat}$. The return type of functions, $\mathtt{R}$, is fixed and is *not* a first-class type. Intuitively, we consider that functions are not expected to return. A type in direct style $A \to B$ becomes in ECPS: $(A, B{\to}\mathtt{R}){\to}\mathtt{R}$. In the typing context $(\Gamma, x : A)$, the free variable x does not appear in Γ.

First, consider the pure fragment of ECPS, without effects, named CPS:

$$\begin{array}{lll}
\text{Values} & v, w := \mathtt{zero} \mid \mathtt{succ}(v) \mid \lambda(x_1{:}A_1, \ldots, x_n{:}A_n).t \mid x & (n \geq 0) \\
\text{Computations} & s, t := v(w_1, \ldots, w_n) \mid \mathtt{case}\ v\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} \mid & \\
 & \quad (\mathtt{rec}\ x.v)(w_1, \ldots, w_n). &
\end{array}$$

Variables, natural numbers and lambdas are values. Computations include function application and an eliminator for natural numbers. The expression $\mathtt{rec}\ x.v$ is a recursive definition of the function v, which must be applied. If exactly one argument appears in a lambda abstraction or an application term, we will sometimes omit the parentheses around that argument.

There are two typing relations in CPS, one for values $\Gamma \vdash v : A$, which says that value v has type A in the context Γ, and one for computations $\Gamma \vdash t : \mathtt{R}$. This says that t is well-formed given the context Γ. All computations have the same return type $\mathtt{R}$. We also define the *order of a type* recursively, which roughly speaking counts the number of function arrows $\to$ in a type.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, \overrightarrow{x : A} \vdash t : \mathtt{R}}{\Gamma \vdash \lambda(\overrightarrow{x{:}A}).t : (\overrightarrow{A}){\to}\mathtt{R}} \qquad \frac{}{\Gamma \vdash \mathtt{zero} : \mathtt{nat}} \qquad \frac{\Gamma \vdash v : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(v) : \mathtt{nat}}$$

$$\frac{\Gamma \vdash v : (\overrightarrow{A}){\to}\mathtt{R} \quad (\Gamma \vdash w_i : A_i)_i}{\Gamma \vdash v\ (\overrightarrow{w}) : \mathtt{R}} \qquad \frac{\Gamma, x : (\overrightarrow{A}){\to}\mathtt{R} \vdash v : (\overrightarrow{A}){\to}\mathtt{R} \quad (\Gamma \vdash w_i : A_i)_i}{\Gamma \vdash (\mathtt{rec}\ x.v)(\overrightarrow{w}) : \mathtt{R}}$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \qquad \Gamma \vdash t : \mathtt{R} \qquad \Gamma, x : \mathtt{nat} \vdash s : \mathtt{R}}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow t,\ \mathtt{succ}(x) \Rightarrow s\} : \mathtt{R}}$$

$$ord(\mathtt{nat}) = 0 \qquad\qquad ord(() {\to} \mathtt{R}) = 1$$

$$ord((A_1, \ldots, A_n){\to}\mathtt{R}) = max_{1 \leq i \leq n}(ord(A_i)) + 1 \qquad \text{(if } n > 0\text{)}$$

To introduce algebraic effects into our language, we consider a new kind of context Σ, disjoint from Γ, which we call an *effect context.* The symbols σ appearing in Σ stand for effect operations and their type must have either order 1 or 2. For example, the binary choice operation $or : (() {\to} \mathtt{R},\ () {\to} \mathtt{R}) {\to} \mathtt{R}$ expects two thunked computations. The output operation $output : (\mathtt{nat},\ () {\to} \mathtt{R}) {\to} \mathtt{R}$ expects a parameter and a continuation. An operation signifying success, which takes no arguments, is $\downarrow\, : () {\to} \mathtt{R}$. Roughly, Σ could be regarded as a countable algebraic signature.

We extend the syntax of CPS with effectful computations. The typing relations now carry a Σ context: $\Gamma \vdash_\Sigma v : A$ and $\Gamma \vdash_\Sigma t : \mathtt{R}$. Otherwise, the typing judgements remain unchanged; we have a new rule for typing effect operations:

$$s, t \coloneqq \ldots \mid \sigma(\overrightarrow{v}, \overrightarrow{k}) \qquad \frac{\sigma : (\overrightarrow{A}, \overrightarrow{B}) {\to} \mathtt{R} \in \Sigma \quad (\Gamma \vdash_\Sigma v_i : A_i)_i \quad (\Gamma \vdash_\Sigma k_j : B_j)_j}{\Gamma \vdash_\Sigma \sigma(\overrightarrow{v}, \overrightarrow{k}) : \mathtt{R}}$$

In ECPS, the only type with order 0 is $\mathtt{nat}$, so in fact $A_i = \mathtt{nat}$ for all i. Notice that the grammar does not allow function abstraction over a symbol from Σ and that σ is not a first-class term. So we can assume that Σ is fixed, as in the examples from Sect. 2.1.

As usual, we identify terms up to alpha-equivalence. Substitution of values for free variables that are not operations, $v[w/x]$ and $t[w/x]$, is defined in the standard way by induction on the structure of v and t. We use $\overline{n}$ to denote the term $\mathtt{succ}^n(\mathtt{zero})$. Let $(\vdash_\Sigma)$ be the set of well-formed closed computations and $(\vdash_\Sigma A)$ the set of closed values of type A.

2.1 Operational Semantics

We define a family of relations on closed computation terms $(\longrightarrow) \subseteq (\vdash_\Sigma) \times (\vdash_\Sigma)$ for any effect context Σ:

$$(\lambda(\overrightarrow{x{:}A}).t)\ (\overrightarrow{w}) \longrightarrow t[\overrightarrow{w}/\overrightarrow{x}]$$
$$(\mathtt{rec}\ x.v)\ (\overrightarrow{w}) \longrightarrow (v[(\lambda(\overrightarrow{y{:}A}).(\mathtt{rec}\ x.v)(\overrightarrow{y}))/x])\ (\overrightarrow{w})$$
$$\mathtt{case\ zero\ of}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} \longrightarrow s$$
$$\mathtt{case\ succ}(v)\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow s,\ \mathtt{succ}(x) \Rightarrow t\} \longrightarrow t[v/x].$$

Observe that the reduction given by $\longrightarrow$ can either run forever or terminate with an effect operation. If the effect operation does not take any arguments of order 1 (i.e. continuations), the computation stops. If the reduction reaches $\sigma(\overrightarrow{v}, \overrightarrow{k})$, the intuition is that any continuation k_i may be chosen, and executed with the results of operation σ. Thus, repeatedly evaluating effect operations leads to the construction of an infinitely branching tree (similar to that in [32]), as we now explain, which we call an *effect tree*. A path in the tree represents a possible execution path of the program.

An effect tree, of possibly infinite depth and width, can contain:

- leaves labelled $\bot$, which signifies nontermination of $\longrightarrow$;
- leaves labelled $\sigma_{\overrightarrow{v}}$, where $\sigma : (\overrightarrow{A}){\to}\mathtt{R} \in \Sigma$ and $(\vdash_\Sigma v_i : A_i)_i$;
- nodes labelled $\sigma_{\overrightarrow{v}}$, where $\sigma : (\overrightarrow{A}, \overrightarrow{B}){\to}\mathtt{R} \in \Sigma$ and each $\vdash_\Sigma v_i : A_i$; such a node has an infinite number of children $t_0, t_1, \ldots$.

Denote the set of all effect trees by Trees_Σ. This set has a partial order: $tr_1 \leq tr_2$ if and only if tr_1 can be obtained by replacing subtrees of tr_2 by $\bot$. Every ascending chain $t_1 \leq t_2 \leq \ldots$ has a least upper bound $\bigsqcup_n t_n$. In fact Trees_Σ is the free pointed Σ-algebra [2] and therefore also has a coinductive property [9].

Next, we define a sequence of effect trees associated with each well-formed closed computation. Each element in the sequence can be seen as evaluating the computation one step further. Let $[\![-]\!]_{(-)} : (\vdash_\Sigma) \times \mathbb{N} \longrightarrow \mathit{Trees}_\Sigma$:

$$[\![t]\!]_0 = \bot$$

$$[\![t]\!]_{m+1} = \begin{cases} [\![s]\!]_m & \text{if } t \longrightarrow s \\ \sigma_{\overrightarrow{v}}\Big(\Big(\big([\![k_i\ (\overline{n_1},\dots,\overline{n_{p_i}})]\!]_m\big)_{n_1,\dots,n_{p_i}\in\mathbb{N}}\Big)_i\Big) & \text{if } t = \sigma(\overrightarrow{v}, \overrightarrow{k}) \end{cases}$$

These are all the cases since well-formed computations do not get stuck. We define the function $[\![-]\!] : (\vdash_\Sigma) \longrightarrow \mathit{Trees}_\Sigma$ as the least upper bound of the chain $\{[\![t_n]\!]\}_{n\in\mathbb{N}}$: $[\![t]\!] = \bigsqcup_{n\in\mathbb{N}} [\![t]\!]_n$.

We now give examples of effect contexts Σ for different algebraic effects, and of some computations and their associated effect trees.

Example 1 (Pure functional computation). $\Sigma = \{\downarrow : ()\to\mathtt{R}\}$. Intuitively, $\downarrow$ is a top-level success flag, analogous to a 'barb' in process algebra. This is to ensure a reasonable contextual equivalence for CPS programs, which never actually return results. For example, $\mathit{loop} = (\mathtt{rec}\ f.\lambda().(f\ x))\ ()$ runs forever, and

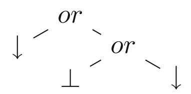
$$\mathtt{test\_zero} = \lambda(y{:}\mathtt{nat}).\ \mathtt{case}\ y\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow\ \downarrow (),\ \mathtt{succ}(x) \Rightarrow \mathit{loop}\}$$

is a continuation that succeeds just when it is passed zero. Generally, an effect tree for a pure computation is either $\downarrow$ if it succeeds or $\bot$ otherwise.

Example 2 (Nondeterminism). $\Sigma = \{\mathit{or} : (()\to\mathtt{R},\ ()\to\mathtt{R})\to\mathtt{R},\ \downarrow : ()\to\mathtt{R}\}$. Intuitively, $\mathit{or}(k_1, k_2)$ performs a nondeterministic choice between computations $k_1\ ()$ and $k_2\ ()$. Consider a continuation $\mathtt{test\_3} : \mathtt{nat}\to\mathtt{R}$ that diverges on 3 and succeeds otherwise. The program $\mathtt{or\_succ}$ from the introduction is in ECPS:

$$\begin{aligned} \mathtt{or\_succ} = \lambda(x{:}\mathtt{nat}, k{:}\mathtt{nat}\to\mathtt{R}).\ &\mathit{or}(\lambda().\ k\ x, \\ &\lambda().\ \mathit{or}(\lambda().k\ (\mathtt{succ}(x)), \\ &\qquad\qquad \lambda().k\ (\mathtt{succ}(\mathtt{succ}(x)))))) \end{aligned}$$

$[\![\mathtt{or\_succ}\ (\overline{2},\ \mathtt{test\_3})]\!] =$

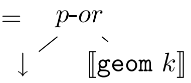


Example 3 (Probabilistic choice). $\Sigma = \{\mathit{p\text{-}or} : (()\to\mathtt{R},\ ()\to\mathtt{R})\to\mathtt{R},\ \downarrow : ()\to\mathtt{R}\}$. Intuitively, the operation $\mathit{p\text{-}or}(k_1, k_2)$ chooses between $k_1\ ()$ and $k_2\ ()$ with probability 0.5. Consider the following term which encodes the geometric distribution:

$$\begin{aligned} &\mathtt{geom} = \lambda k{:}\mathtt{nat}\to\mathtt{R}. \\ &\quad \big(\mathtt{rec}\ f.\ \lambda(n{:}\mathtt{nat}, k'{:}\mathtt{nat}\to\mathtt{R}).\mathit{p\text{-}or}(\lambda().k'\ n,\ \lambda().f\ (\mathtt{succ}(n), k'))\big)\ (\overline{1}, k). \end{aligned}$$

The probability that $\mathtt{geom}$ passes a number $n > 0$ to its continuation is 2^{-n}. To test it, consider $k = (\lambda x{:}\mathtt{nat}.\ \downarrow ())$; then $[\![\mathtt{geom}\ k]\!]$ is an infinite tree:

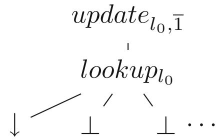
$[\![\mathtt{geom}\ k]\!] =$



Example 4 (Global store). $\mathbb{L}$ is a finite set of locations storing natural numbers and $\Sigma = \{\mathit{lookup}_l : (\mathtt{nat}\to\mathtt{R})\to\mathtt{R},\ \mathit{update}_l : (\mathtt{nat}, ()\to\mathtt{R})\to\mathtt{R} \mid l \in \mathbb{L}\} \cup \{\downarrow : ()\to\mathtt{R}\}$. Intuitively, $\mathit{lookup}_l(k)$ looks up the value at storage location l, if this is $\overline{n}$ it

continues with $k\,(\overline{n})$. For $\mathit{update}_l(v, k)$ the intuition is: write the number v in location l then continue with the computation $k\,()$. For example:

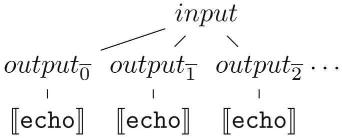
$$[\![\mathit{update}_{l_0}(\overline{1},\ \lambda().\mathit{lookup}_{l_0}(\lambda x{:}\mathtt{nat}.\mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow \downarrow\ (),\ \mathtt{succ}(y) \Rightarrow \mathit{loop}\}))]\!] =$$



Only the second branch of lookup_{l_0} can occur. The other branches are still present in the tree because $[\![-]\!]$ treats effect operations as uninterpreted syntax.

Example 5 (Interactive input/output). $\Sigma = \{\downarrow : ()\to\mathtt{R},\ \mathit{output} : (\mathtt{nat}, ()\to\mathtt{R})\to\mathtt{R},\ \mathit{input} : (\mathtt{nat}\to\mathtt{R})\to\mathtt{R}\}$. Intuitively, the computation $\mathit{input}(k)$ accepts number $\overline{n}$ from the input channel and continues with $k\,(\overline{n})$. The computation $\mathit{output}(v, k)$ writes v to the output channel then continues with computation $k\,()$. Below is a computation that inputs a number $\overline{n}$ then outputs it immediately, and repeats.

$$[\![\mathtt{echo}]\!] = [\![\big(\mathtt{rec}\ f.\ \lambda().\ \mathit{input}(\lambda x{:}\mathtt{nat}.\ \mathit{output}(x,\ \lambda().f\ ()))\big)\ ()]\!] =$$


input
$\mathit{output}_{\overline{0}}$
$\mathit{output}_{\overline{1}}$
$\mathit{output}_{\overline{2}} \cdots$
$[\![\mathtt{echo}]\!]$
$[\![\mathtt{echo}]\!]$
$[\![\mathtt{echo}]\!]$

2.2 Contextual Equivalence

Informally, two terms are contextually equivalent if they have the same *observable behaviour* in all program contexts. The definition of observable behaviour depends on the programming language under consideration. In ECPS, we can observe effectful behaviour such as interactive output values or the probability with which a computation succeeds. This behaviour is encoded by the effect tree of a computation. Therefore, we represent an ECPS observation as a set of effect trees P. A computation t exhibits observation P if $[\![t]\!] \in P$.

For a fixed set of effect operations Σ, we define the set $\mathfrak{P}$ of possible *observations.* The elements of $\mathfrak{P}$ are subsets of Trees_Σ. Observations play a similar role to the modalities from [38]. For our running examples of effects, $\mathfrak{P}$ is defined as follows:

Example 6 (Pure functional computation). Define $\mathfrak{P} = \{\Downarrow\}$ where $\Downarrow = \{\downarrow\}$. There are no effect operations so the $\Downarrow$ observation only checks for success.

Example 7 (Nondeterminism). Define $\mathfrak{P} = \{\Diamond, \Box\}$ where:

$$\Diamond = \{tr \in \mathit{Trees}_\Sigma \mid \text{at least one of the paths in } tr \text{ has a } \downarrow \text{ leaf}\}$$
$$\Box = \{tr \in \mathit{Trees}_\Sigma \mid \text{the paths in } tr \text{ are all finite and finish with a } \downarrow\}.$$

The intuition is that, if $[\![t]\!] \in \Diamond$, then computation t *may* succeed, whereas if $[\![t]\!] \in \Box$, then t *must* succeed.

Example 8 (Probabilistic choice). Define $\mathbb{P} : \mathit{Trees}_\Sigma \longrightarrow [0,1]$ to be the least function, by the pointwise order, such that:

$$\mathbb{P}(\downarrow) = 1 \qquad \mathbb{P}(\mathit{p\text{-}or}(\mathit{tr}_0, \mathit{tr}_1)) = \frac{1}{2}\mathbb{P}(\mathit{tr}_0) + \frac{1}{2}\mathbb{P}(\mathit{tr}_1).$$

Notice that $\mathbb{P}(\bot) = 0$. Then observations are defined as:

$$\mathbf{P}_{>q} = \{\mathit{tr} \in \mathit{Trees}_\Sigma \mid \mathbb{P}(\mathit{tr}) > q\} \qquad \mathfrak{P} = \{\mathbf{P}_{>q} \mid q \in \mathbb{Q},\ 0 \leq q < 1\}.$$

This means that $[\![t]\!] \in \mathbf{P}_{>q}$ if the probability that t succeeds is greater than q.

Example 9 (Global store). Define the set of states as the set of functions from storage locations to natural numbers: $\mathit{State} = \mathbb{L} \longrightarrow \mathbb{N}$. Given a state S, we write $[S\downarrow] \subseteq \mathit{Trees}_\Sigma$ for the set of effect trees that terminate when starting in state S. More precisely, $[-]$ is the least State-indexed family of sets satisfying the following:

$$\frac{-}{\downarrow \in [S\downarrow]} \qquad \frac{l \in \mathbb{L} \qquad \mathit{tr}_{S(l)} \in [S\downarrow]}{\mathit{lookup}_l(\mathit{tr}_0, \mathit{tr}_1, \mathit{tr}_2, \ldots) \in [S\downarrow]} \qquad \frac{l \in \mathbb{L} \qquad \mathit{tr} \in [S[l := n]\downarrow]}{\mathit{update}_{l,\overline{n}}(\mathit{tr}) \in [S\downarrow]}$$

The set of observations is: $\mathfrak{P} = \{[S\downarrow] \mid S \in \mathit{State}\}$.

Example 10 (Interactive input/output). An I/O-trace is a finite word w over the alphabet $\{?n \mid n \in \mathbb{N}\} \cup \{!n \mid n \in \mathbb{N}\}$. For example, $?1\,!1\,?2\,!2\,?3\,!3$. The set of observations is: $\mathfrak{P} = \{\langle W\rangle_{\ldots},\ \langle W\rangle{\downarrow} \mid W \text{ an I/O-trace}\}$. Observations are defined as the least sets satisfying the following rules:

$$\frac{-}{\mathit{tr} \in \langle\epsilon\rangle_{\ldots}} \quad \frac{\mathit{tr} = \downarrow}{\mathit{tr} \in \langle\epsilon\rangle{\downarrow}} \quad \frac{\mathit{tr}_n \in \langle W\rangle_{\ldots}}{\mathit{input}(\mathit{tr}_0, \mathit{tr}_1, \ldots) \in \langle(?n)W\rangle_{\ldots}} \quad \frac{\mathit{tr}' \in \langle W\rangle_{\ldots}}{\mathit{output}_{\overline{n}}(\mathit{tr}') \in \langle(!n)W\rangle_{\ldots}}$$

and the analogous rules for $\langle(?n)W\rangle{\downarrow}$ and $\langle(!n)W\rangle{\downarrow}$. Thus, $[\![t]\!] \in \langle W\rangle_{\ldots}$ if computation t produces I/O trace W, and $[\![t]\!] \in \langle W\rangle{\downarrow}$ if additionally t succeeds immediately after producing W.

Using the set of observations $\mathfrak{P}$, we can now define contextual equivalence as the greatest compatible and adequate equivalence relation between possibly open terms of the same type. Adequacy specifies a necessary condition for two *closed* computations to be related, namely producing the same observations.

Definition 1. *A well-typed relation $\mathcal{R} = (\mathcal{R}^{\mathfrak{v}}_A, \mathcal{R}^{\mathfrak{c}})$ (i.e. a family of relations indexed by ECPS types where $\mathcal{R}^{\mathfrak{c}}$ relates computations) on possibly open terms is adequate if:*

$$\forall s, t.\ \vdash_\Sigma s\ \mathcal{R}^{\mathfrak{c}}\ t \implies \forall P \in \mathfrak{P}.\ [\![s]\!] \in P \iff [\![t]\!] \in P.$$

Relation $\mathcal{R}$ is compatible if it is closed under the rules in [21, Page 57]. As an example, the rules for application and lambda abstraction are:

$$\frac{\Gamma \vdash_\Sigma v\ \mathcal{R}^{\mathfrak{v}}_{(\overrightarrow{A})\to\mathtt{R}}\ v' \qquad (\Gamma \vdash_\Sigma w_i\ \mathcal{R}^{\mathfrak{v}}_{A_i}\ w'_i)_i}{\Gamma \vdash_\Sigma v(\overrightarrow{w})\ \mathcal{R}^{\mathfrak{c}}\ v'(\overrightarrow{w'})} \qquad \frac{\Gamma, \overrightarrow{x : A} \vdash_\Sigma s\ \mathcal{R}^{\mathfrak{c}}\ t}{\Gamma \vdash_\Sigma \lambda(\overrightarrow{x{:}A}).s\ \mathcal{R}^{\mathfrak{v}}_{(\overrightarrow{A})\to\mathtt{R}}\ \lambda(\overrightarrow{x{:}A}).t}$$

Definition 2 (Contextual equivalence). *Let $\mathbb{CA}$ be the set of well-typed relations on possibly open terms that are both compatible and adequate. Define contextual equivalence $\equiv_{\text{ctx}}$ to be $\bigcup \mathbb{CA}$.*

Proposition 1. *Contextual equivalence $\equiv_{\text{ctx}}$ is an equivalence relation, and is moreover compatible and adequate.*

This definition of contextual equivalence, originally proposed in [11,19], can be easily proved equivalent to the traditional definition involving program contexts (see [21, §7]). As Pitts observes [30], reasoning about program contexts directly is inconvenient because they cannot be defined up to alpha-equivalence, hence we prefer using Definition 2.

For example, in the pure setting (Example 1), we have $\overline{0} \not\equiv_{\text{ctx}} \overline{1}$, because $\texttt{test\_zero}(\overline{0}) \not\equiv_{\text{ctx}} \texttt{test\_zero}(\overline{1})$; they are distinguished by the observation $\Downarrow$. In the state example, $\mathit{lookup}_{l_1}(k) \not\equiv_{\text{ctx}} \mathit{lookup}_{l_2}(k)$, because they are distinguished by the context $(\lambda k{:}\texttt{nat}{\to}\texttt{R}.\,[-])\ (\texttt{test\_zero})$ and the observation $[S{\downarrow}]$ where $S(l_1) = \overline{0}$ and $S(l_2) = \overline{1}$. In the case of probabilistic choice (Example 3), $\texttt{geom}\ (\lambda x{:}\texttt{nat}.\downarrow\ ()) \equiv_{\text{ctx}} \downarrow()$ because $(\texttt{geom}\ (\lambda x{:}\texttt{nat}.\downarrow\ ()))$ succeeds with probability 1 (‘almost surely’).

3 A Program Logic for ECPS – $\mathcal{F}$

This section contains the main contribution of the paper: a logic $\mathcal{F}$ of program properties for ECPS which characterizes contextual equivalence. Crucially, the logic makes use of the observations in $\mathfrak{P}$ to express properties of computations.

In $\mathcal{F}$, there is a distinction between formulas that describe values and those that describe computations. Each value formula is associated an ECPS type A. Value formulas are constructed from the basic formulas $(\phi_1, \ldots, \phi_n) \mapsto P$ and $\phi = \{n\}$, where $n \in \mathbb{N}$ and $P \in \mathfrak{P}$, as below. The indexing set I can be infinite, even uncountable. Computation formulas are simply the elements of $\mathfrak{P}$.

$$\frac{n \in \mathbb{N}}{\{n\} : \texttt{nat}} \qquad \overset{(\text{VAL})}{\frac{\phi_1 : A_1 \ldots \phi_n : A_n \qquad P \in \mathfrak{P}}{(\phi_1, \ldots, \phi_n) \mapsto P : (A_1, \ldots, A_n) {\to} \texttt{R}}} \qquad \frac{(\phi_i : A)_{i \in I}}{\vee_{i \in I} \phi_i : A} \qquad \frac{(\phi_i : A)_{i \in I}}{\wedge_{i \in I} \phi_i : A} \qquad \frac{\phi : A}{\neg \phi : A}$$

The satisfaction relation $\models_{\mathcal{F}}$ relates a closed value $\vdash_{\Sigma} v : A$ to a value formula $\phi : A$ of the same type, or a closed computation t to an observation P. Relation $t \models_{\mathcal{F}} P$ tests the shape of the effect tree of t.

$$\begin{array}{rcl}
v \models_{\mathcal{F}} \{n\} & \iff & v = \overline{n} \\
v \models_{\mathcal{F}} (\phi_1, \ldots, \phi_n) \mapsto P & \iff & \text{for all closed values } w_1, \ldots, w_n \text{ such that} \\
 & & \forall i.\ w_i \models_{\mathcal{F}} \phi_i \text{ then } v(w_1, \ldots, w_n) \models_{\mathcal{F}} P \\
v \models_{\mathcal{F}} \vee_{i \in I} \phi_i & \iff & \text{there exists } j \in I \text{ such that } v \models_{\mathcal{F}} \phi_j \\
v \models_{\mathcal{F}} \wedge_{i \in I} \phi_i & \iff & \text{for all } j \in I,\ v \models_{\mathcal{F}} \phi_j \\
v \models_{\mathcal{F}} \neg \phi & \iff & \text{it is false that } v \models_{\mathcal{F}} \phi \\
t \models_{\mathcal{F}} P & \iff & [\![t]\!] \in P.
\end{array}$$

Example 11. Consider the following formulas, where only ϕ_3 and ϕ_4 refer to the same effect context:

$$\begin{aligned}
\phi_1 &= \big((\{3\} \mapsto \Diamond) \mapsto \Diamond\big) \wedge \big((\{4\} \mapsto \Diamond) \mapsto \Diamond\big) \wedge \big((\{3\} \mapsto \Box \wedge \{4\} \mapsto \Box) \mapsto \Box\big)\\
\phi_2 &= ((\vee_{n>1}\{n\}) \mapsto \mathbf{P}_{>q}) \mapsto \mathbf{P}_{>q/2}\\
\phi_3 &= \wedge_{S \in \mathit{State}}\big((\{S(l)\} \mapsto [S{\downarrow}]) \mapsto [S{\downarrow}]\big)\\
\phi_4 &= \wedge_{S \in \mathit{State}} \wedge_{n \in \mathbb{N}} \big((\{n\},\ () \mapsto [S[l_0 \coloneqq n, l_1 \coloneqq n+1]{\downarrow}]) \mapsto [S[l_0 \coloneqq n]{\downarrow}]\big)\\
\phi_5 &= \wedge_{k \in \mathbb{N}} \vee_{n_1,\ldots,n_k \in \mathbb{N}} (() \mapsto \langle ?n_1!n_1?n_2!n_2\ \ldots\ ?n_k!n_k\rangle_{\ldots}).
\end{aligned}$$

Given a function $v : (\texttt{nat}{\to}\texttt{R}){\to}\texttt{R}$, $v \models_{\mathcal{F}} \phi_1$ means that v is guaranteed to call its argument only with $\overline{3}$ or $\overline{4}$. The function `geom` from Example 3 satisfies ϕ_2 because with probability 1/2 it passes to the continuation a number $n > 1$.

For example, the following satisfactions hold: $\lambda k{:}\texttt{nat}{\to}\texttt{R}.\, \mathit{lookup}_l(k) \models_{\mathcal{F}} \phi_3$ and $f = \lambda(x{:}\texttt{nat}, k{:}(){\to}\texttt{R}).\, \mathit{update}_{l_1}(\texttt{succ}(x), k) \models_{\mathcal{F}} \phi_4$. The latter formula says that, either f always succeeds, or f evaluated with $\overline{n}$ changes the state from $S[l_0 \coloneqq n]$ to $S[l_0 \coloneqq n, l_1 \coloneqq n+1]$ before calling its continuation. This is similar to a total correctness assertion $[S[l_0 \coloneqq n]](-)[S[l_0 \coloneqq n, l_1 \coloneqq n+1]]$ from Hoare logic, for a direct style program. Formula ϕ_5 is satisfied by $\lambda().\texttt{echo}$, where `echo` is the computation defined in Example 5.

Even though the indexing set I in $\wedge_{i \in I}$ and $\vee_{i \in I}$ may be uncountable, the sets of values and computations are countable. Since logical formulas are interpreted over values and computations, all conjunctions and disjunctions are logically equivalent to countable ones.

Definition 3 (Logical equivalence). *For any closed values $\vdash_{\Sigma} v_1 : A$ and $\vdash_{\Sigma} v_2 : A$, and for any closed computations $\vdash_{\Sigma} s_1$ and $\vdash_{\Sigma} s_2$:*

$$\begin{aligned}
v_1 \equiv_{\mathcal{F}} v_2 &\iff \forall \phi : A \textit{ in } \mathcal{F}.\ (v_1 \models_{\mathcal{F}} \phi \iff v_2 \models_{\mathcal{F}} \phi)\\
s_1 \equiv_{\mathcal{F}} s_2 &\iff \forall P \textit{ in } \mathcal{F}.\ (s_1 \models_{\mathcal{F}} P \iff s_2 \models_{\mathcal{F}} P).
\end{aligned}$$

To facilitate equational reasoning, logical equivalence should be compatible, a property proved in the next section (Proposition 3). Compatibility allows substitution of related programs for a free variable that appears on both sides of a program equation. Notice that logical equivalence would not be changed if we added conjunction, disjunction and negation at the level of computation formulas. We have omitted such connectives for simplicity.

To state our main theorem, first define the open extension of a well-typed relation $\mathcal{R}$ on closed terms as: $\overrightarrow{x : A} \vdash_{\Sigma} t\ \mathcal{R}^{\circ}\ s$ if and only if for any closed values $(\vdash_{\Sigma} v_i : A_i)_i$, $t[\overrightarrow{v/x}]\ \mathcal{R}\ s[\overrightarrow{v/x}]$. Three sufficient conditions that we impose on the set of observations $\mathfrak{P}$ are defined below. The first one, consistency, ensures that contextual equivalence can distinguish at least two programs.

Definition 4 (Consistency). *A set of observations $\mathfrak{P}$ is consistent if there exists at least one observation $P_0 \in \mathfrak{P}$ such that:*

1. $P_0 \neq \mathit{Trees}_\Sigma$ and
2. there exists at least one computation t_0 such that $[\![t_0]\!] \in P_0$.

Definition 5 (Scott-openness). *A set of trees X is Scott-open if:*

1. It is upwards closed, that is: $tr \in X$ and $tr \leq tr'$ imply $tr' \in X$.
2. Whenever $tr_1 \leq tr_2 \leq \ldots$ is an ascending chain with least upper bound $\bigsqcup tr_i \in X$, then $tr_j \in X$ for some j.

Definition 6 (Decomposability). *The set of observations $\mathfrak{P}$ is decomposable if for any $P \in \mathfrak{P}$, and for any $tr \in P$:*

$$\forall \sigma \in \Sigma.\ \big(tr = \sigma_{\overrightarrow{v}}(\overrightarrow{tr'}) \implies \exists \overrightarrow{P'} \in \mathfrak{P} \cup \{\mathit{Trees}_\Sigma\}.\ \overrightarrow{tr'} \in \overrightarrow{P'} \text{ and } \forall \overrightarrow{p'} \in \overrightarrow{P'}.\ \sigma_{\overrightarrow{v}}(\overrightarrow{p'}) \in P\big).$$

Theorem 1 (Soundness and Completeness of $\mathcal{F}$). *For a decomposable set of Scott-open observations $\mathfrak{P}$ that is consistent, the open extension of $\mathcal{F}$-logical equivalence coincides with contextual equivalence: $(\equiv_{\mathcal{F}}^{\circ}) = (\equiv_{\mathrm{ctx}})$.*

The proof of this theorem is outlined in Sect. 4. It is easy to see that for all running examples of effects the set $\mathfrak{P}$ is consistent. The proof that each $P \in \mathfrak{P}$ is Scott-open is similar to that for modalities from [38]. It remains to show that for all our examples $\mathfrak{P}$ is decomposable. Intuitively, decomposability can be understood as saying that logical equivalence is a congruence for the effect context Σ.

Example 12 (Pure functional computation). The only observation is $\Downarrow = \{\downarrow\}$. There are no trees in $\Downarrow$ whose root has children, so decomposability is satisfied.

Example 13 (Nondeterminism). Consider $tr \in \Diamond$. Either $tr = \downarrow$, in which case we are done, or $tr = or(tr'_0, tr'_1)$. It must be the case that either tr'_0 or tr'_1 have a $\downarrow$-leaf. Without loss of generality, assume this is the case for tr'_0. Then we know $tr'_0 \in \Diamond$ so we can choose $P'_0 = \Diamond, P'_1 = \mathit{Trees}_\Sigma$. For any $\overrightarrow{p'} \in \overrightarrow{P'}$ we know $or(\overrightarrow{p'}) \in \Diamond$ because p'_0 has a $\downarrow$-leaf, so decomposability holds. The argument for $tr \in \Box$ is analogous: $P'_0 = P'_1 = \Box$.

Example 14 (Probabilistic choice). Consider $tr = p\text{-}or(tr'_0, tr'_1) \in \mathbf{P}_{>q}$. Choose: $q_0 = \frac{\mathbb{P}(tr'_0)}{\mathbb{P}(tr'_0)+\mathbb{P}(tr'_1)} \cdot 2q$ and $q_1 = \frac{\mathbb{P}(tr'_1)}{\mathbb{P}(tr'_0)+\mathbb{P}(tr'_1)} \cdot 2q$. From $\mathbb{P}(tr) = \frac{1}{2}(\mathbb{P}(tr'_0)+\mathbb{P}(tr'_1)) > q$ we can deduce that: $1 \geq \mathbb{P}(tr'_0) > q_0$ and $1 \geq \mathbb{P}(tr'_1) > q_1$. So we can choose $P'_0 = \mathbf{P}_{>q_0}, P'_1 = \mathbf{P}_{>q_1}$ to satisfy decomposability.

Example 15 (Global store). Consider a tree $tr = \sigma_{\overrightarrow{v}}(tr'_0, tr'_1, tr'_2, \ldots) \in [S{\downarrow}]$. If $\sigma = \mathit{lookup}_l$, then we know $tr'_{S(l)} \in [S{\downarrow}]$. In the definition of decomposability, choose $P'_{S(l)} = [S{\downarrow}]$ and $P'_{k \neq S(l)} = \mathit{Trees}_\Sigma$ and we are done. If $\sigma_{\overrightarrow{v}} = \mathit{update}_{l,\overline{n}}$, then $tr'_0 \in [S[l \coloneqq n]{\downarrow}]$. Choose $P'_0 = [S[l \coloneqq n]{\downarrow}]$.

Example 16 (Interactive input/output). Consider an I/O trace $W \neq \epsilon$ and a tree $tr = \sigma_{\overrightarrow{v}}(tr'_0, tr'_1, tr'_2, \ldots) \in \langle W \rangle_{\ldots}$. If $\sigma = input$, it must be the case that $W = (?k)W'$ and $tr'_k \in \langle W' \rangle_{\ldots}$. We can choose $P'_k = \langle W' \rangle_{\ldots}$ and $P'_{m \neq k} = \langle \epsilon \rangle_{\ldots}$ and we are done. If $\sigma_{\overrightarrow{v}} = output_{\overline{n}}$, then $W = (!n)W'$ and $tr'_0 \in \langle W' \rangle_{\ldots}$. Choose $P'_0 = \langle W' \rangle_{\ldots}$ and we are done. The proof for $\langle W \rangle\downarrow$ is analogous.

4 Soundness and Completeness of the Logic $\mathcal{F}$

This section outlines the proof of Theorem 1, which says that $\mathcal{F}$-logical equivalence coincides with contextual equivalence. The full proof can be found in [21]. First, we define applicative bisimilarity for ECPS, similarly to the way Simpson and Voorneveld [38] define it for PCF with algebraic effects. Then, we prove in turn that $\mathcal{F}$-logical equivalence coincides with applicative bisimilarity, and that applicative bisimilarity coincides with contextual equivalence. Thus, three notions of program equivalence for ECPS are in fact the same.

Definition 7 (Applicative $\mathfrak{P}$-bisimilarity). *A collection of relations* $\mathcal{R}^{\mathfrak{v}}_A \subseteq (\vdash_\Sigma A)^2$ *for each type* A *and* $\mathcal{R}^{\mathfrak{c}} \subseteq (\vdash_\Sigma)^2$ *is an applicative* $\mathfrak{P}$-simulation *if:*

1. $v \, \mathcal{R}^{\mathfrak{v}}_{\mathtt{nat}} \, w \implies v = w.$
2. $s \, \mathcal{R}^{\mathfrak{c}} \, t \implies \forall P \in \mathfrak{P}.\ (\llbracket s \rrbracket \in P \implies \llbracket t \rrbracket \in P).$
3. $v \, \mathcal{R}^{\mathfrak{v}}_{(\overrightarrow{A})\to\mathtt{R}} \, u \implies \forall (\vdash_\Sigma w_i : A_i)_i.\ v(\overrightarrow{w}) \, \mathcal{R}^{\mathfrak{c}} \, u(\overrightarrow{w}).$

An applicative $\mathfrak{P}$-bisimulation *is a symmetric simulation.* Bisimilarity*, denoted by* $\sim$*, is the union of all bisimulations. Therefore, it is the greatest applicative* $\mathfrak{P}$*-bisimulation.*

Notice that applicative bisimilarity uses the set of observations $\mathfrak{P}$ to relate computations, just as contextual and logical equivalence do. It is easy to show that bisimilarity is an equivalence relation.

Proposition 2. *Given a decomposable set of Scott-open observations* $\mathfrak{P}$*, the open extension of applicative* $\mathfrak{P}$*-bisimilarity,* $\sim^\circ$*, is compatible.*

Proof (notes). This is proved using Howe's method [14], following the structure of the corresponding proof from [38]. Scott-openness is used to show that the observations P interact well with the sequence of trees $\llbracket - \rrbracket_{(-)}$ associated with each computation. For details see [21, §5.4]. □

Proposition 3. *Given a decomposable set of Scott-open observations* $\mathfrak{P}$*, applicative* $\mathfrak{P}$*-bisimilarity* $\sim$ *coincides with* $\mathcal{F}$*-logical equivalence* $\equiv_\mathcal{F}$*. Hence, the open extension of* $\mathcal{F}$*-logical equivalence* $\equiv^\circ_\mathcal{F}$ *is compatible.*

Proof (sketch). We define a new logic $\mathcal{V}$ which is almost the same as $\mathcal{F}$ except that the (VAL) rule is replaced by:

$$\frac{\vdash_\Sigma w_1 : A_1 \ldots \vdash_\Sigma w_n : A_n \quad P \in \mathfrak{P}}{(w_1, \ldots, w_n) \mapsto P : (A_1, \ldots, A_n)\to\mathtt{R}} \qquad v \models_\mathcal{V} (\overrightarrow{w}) \mapsto P \iff v(\overrightarrow{w}) \models_\mathcal{V} P.$$

That is, formulas of function type are now constructed using ECPS values. It is relatively straightforward to show that $\mathcal{V}$-logical equivalence coincides with applicative $\mathfrak{P}$-bisimilarity [21, Prop. 6.3.1]. However, we do not know of a similar direct proof for the logic $\mathcal{F}$. From Proposition 2, we deduce that $\mathcal{V}$-logical equivalence is compatible.

Next, we prove that the logics $\mathcal{F}$ and $\mathcal{V}$ are in fact equi-expressive, so they induce the same relation of logical equivalence on ECPS programs [21, Prop. 6.3.4]. Define a translation of formulas from $\mathcal{F}$ to $\mathcal{V}$, $(-)^\flat$, and one from $\mathcal{V}$ to $\mathcal{F}$, $(-)^\sharp$. The most interesting cases are those for formulas of function type:

$$((\phi_1, \ldots, \phi_n) \mapsto P)^\flat = \bigwedge \{(w_1, \ldots, w_n) \mapsto P \mid w_1 \models_{\mathcal{V}} \phi_1^\flat, \ldots, w_n \models_{\mathcal{V}} \phi_n^\flat\}$$
$$((w_1, \ldots, w_n) \mapsto P)^\sharp = (\chi_{w_1}, \ldots, \chi_{w_n}) \mapsto P$$

where χ_{w_i} is the characteristic formula of w_i, that is $\chi_{w_i} = \bigwedge \{\phi \mid w_i \models_{\mathcal{F}} \phi\}$. Equi-expressivity means that the satisfaction relation remains unchanged under both translations, for example $v \models_{\mathcal{V}} \phi \iff v \models_{\mathcal{F}} \phi^\sharp$. Most importantly, the proof of equi-expressivity makes use of compatibility of $\equiv_{\mathcal{V}}$, which we established previously. For a full proof see [21, Prop. 6.2.3]). □

Finally, to prove Theorem 1 we show that applicative $\mathfrak{P}$-bisimilarity coincides with contextual equivalence [21, Prop. 7.2.2]:

Proposition 4. *Consider a decomposable set $\mathfrak{P}$ of Scott-open observations that is consistent. The open extension of applicative $\mathfrak{P}$-bisimilarity $\sim^\circ$ coincides with contextual equivalence $\equiv_{\mathrm{ctx}}$.*

Proof (sketch). Prove $(\equiv_{\mathrm{ctx}}) \subseteq (\sim^\circ)$ in two stages: first we show it holds for closed terms by showing $\equiv_{\mathrm{ctx}}$ for them is a bisimulation; we make use of consistency of $\mathfrak{P}$ in the case of natural numbers. Then we extend to open terms using compatibility of $\equiv_{\mathrm{ctx}}$. The opposite inclusion follows immediately by compatibility and adequacy of $\sim^\circ$. □

5 Related Work

The work closest to ours is that by Simpson and Voorneveld [38]. In the context of a direct-style language with algebraic effects, EPCF, they propose a modal logic which characterizes applicative bisimilarity but not contextual equivalence. Consider the following example from [19] (we use simplified EPCF syntax):

$$M = \lambda().?\mathtt{nat} \qquad N = \textbf{let } y \Rightarrow ?\mathtt{nat} \textbf{ in } \lambda().\mathtt{min}(?\mathtt{nat}, \mathtt{y}) \qquad (1)$$

where $?\mathtt{nat}$ is a computation, defined using *or*, which returns a natural number nondeterministically. Term M satisfies the formula $\Phi = \Diamond(\mathit{true} \mapsto \wedge_{n\in\mathbb{N}}\Diamond\{n\})$ in the logic of [38], which says that M may return a function which in turn may return any natural number. However, N does not satisfy Φ because it always returns a *bounded* number generator G. The bound on G is arbitrarily high so M and N are contextually equivalent, since a context can only test a finite number of outcomes of G.

EPCF can be translated into ECPS via a continuation-passing translation that preserves the shape of computation trees. The translation maps a value $\Gamma \vdash V : \tau$ to a value $\Gamma^* \vdash V^* : \tau^*$. An EPCF computation $\Gamma \vdash M : \tau$ becomes an ECPS value $\Gamma^* \vdash M^* : (\tau^* {\to} \mathtt{R}) {\to} \mathtt{R}$, which intuitively is waiting for a continuation k to pass its return result to (see [21, §4]). As an example, consider the cases for functions and application, where k stands for a continuation:

$$(\Gamma \vdash \lambda x{:}\tau.M : \tau \to \rho)^* = \Gamma^* \vdash \lambda(x{:}\tau^*, k{:}\rho^* {\to} \mathtt{R}).\, (M^*\ k) : (\tau^*, (\rho^* {\to} \mathtt{R})) {\to} \mathtt{R}$$
$$(\Gamma \vdash V\ W : \rho)^* = \Gamma^* \vdash \lambda k{:}\rho^* {\to} \mathtt{R}.\, V^*\ (W^*, k) : (\rho^* {\to} \mathtt{R}) {\to} \mathtt{R}.$$

This translation suggests that ECPS functions of type $(A_1, \ldots, A_n) {\to} \mathtt{R}$ can be regarded as continuations that never return. In EPCF the CPS-style algebraic operations can be replaced by direct-style generic effects [34], e.g. $\mathit{input}() : \mathtt{nat}$.

One way to understand this CPS translation is that it arises from the fact that $((-) {\to} \mathtt{R}) {\to} \mathtt{R}$ is a monad on the multicategory of values (in a suitable sense, e.g. [40]), which means that we can use the standard monadic interpretation of a call-by-value language. As usual, the algebraic structure on the return type $\mathtt{R}$ induces an algebraic structure on the entire monad (see e.g. [16], [24, §8]). We have not taken a denotational perspective in this paper, but for the reader with this perspective, a first step is to note that the quotient set $Q \overset{\text{def}}{=} (\mathit{Trees}_\Sigma)/_{\equiv_{\mathfrak{P}}}$ is a Σ-algebra, where $(\mathit{tr} \equiv_{\mathfrak{P}} \mathit{tr}')$ if $\forall P \in \mathfrak{P}$, $(\mathit{tr} \in P \iff \mathit{tr}' \in P)$; decomposability implies that $(\equiv_{\mathfrak{P}})$ is a Σ-congruence. This thus induces a CPS monad $Q^{(Q^-)}$ on the category of cpos.

Note that the terms in (1) above are an example of programs that are not bisimilar in EPCF but become bisimilar when translated to ECPS. This is because in ECPS bisimilarity, like contextual and logical equivalence, uses continuations to test return results. Therefore, in ECPS we cannot test for all natural numbers, like formula Φ does. This example provides an intuition of why we were able to show that all three notions of equivalence coincide, while [38] was not.

The modalities in Simpson's and Voorneveld's logic are similar to the observations from $\mathfrak{P}$, because they also specify shapes of effect trees. Since EPCF computations have a return value, a modality is used to *lift* a formula about the return values to a computation formula. In contrast, in the logic $\mathcal{F}$ observations alone suffice to specify properties of computations. From this point of view, our use of observations is closer to that found in the work of Johann et al. [17]. This use of observations also leads to a much simpler notion of decomposability (Definition 6) than that found in [38].

It can easily be shown that for the running examples of effects, $\mathcal{F}$-logical equivalence induces the program equations which are usually used to axiomatize algebraic effects, for example the equations for global store from [33]. Thus our choice of observations is justified further.

A different logic for algebraic effects was proposed by Plotkin and Pretnar [35]. It has a modality for each effect operation, whereas observations in $\mathfrak{P}$ are determined by the behaviour of effects, rather than by the syntax of their operations. Plotkin and Pretnar prove that their logic is sound for establishing several notions of program equivalence, but not complete in general. Refinement types are yet another approach to specifying the behaviour of algebraic effects, (e.g. [3]). Several monadic-based logics for computational effects have been proposed, such as [10], [29], although without the focus on contextual equivalence.

A logic describing a higher-order language with local store is the Hoare logic of Yoshida, Honda and Berger [42]. Hoare logic has also been integrated into a type system for a higher-order functional language with dependent types, in the form of Hoare type theory [27]. Although we do not yet know how to deal with local state or dependent types in the logic $\mathcal{F}$, an advantage of our logic over the previous two is that we describe different algebraic effects in a uniform manner.

Another aspect worth noticing is that some (non-trivial) $\mathcal{F}$-formulas are not inhabited by any program. For example, there is no function $v : (()\to\mathtt{R})\to\mathtt{R}$ satisfying: $\psi = (() \mapsto \langle !0\rangle_{\ldots}) \mapsto \langle !1\rangle_{\ldots} \ \wedge\ (() \mapsto \langle !1\rangle_{\ldots}) \mapsto \langle !0\rangle_{\ldots}$.

Formula ψ says that, if the first operation of a continuation is $\mathit{output}(\overline{0})$, this is replaced by $\mathit{output}(\overline{1})$ and vice-versa. But in ECPS, one cannot check whether an argument outputs something without also causing the output observation, and so the formula is never satisfied.

However, ψ could be inhabited if we extended ECPS to allow λ-abstraction over the symbols in the effect context Σ, and allowed such symbols to be *captured* during substitution (dynamic scoping). Consider the following example in an imaginary extended ECPS where we abstract over *output*:

$$\begin{aligned} h = \lambda(x{:}\mathtt{nat}, k{:}()\to\mathtt{R}).\ &\mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow \mathit{output}(\overline{1}, k),\ \mathtt{succ}(y) \Rightarrow \\ &\mathtt{case}\ y\ \mathtt{of}\ \{\mathtt{zero} \Rightarrow \mathit{output}(\overline{0}, k),\ \mathtt{succ}(z) \Rightarrow k\ ()\}\} \\ v = \lambda f{:}()\to\mathtt{R}.&\big((\lambda \mathit{output}{:}(\mathtt{nat}, ()\to\mathtt{R})\to\mathtt{R}.\ (f\ ()))\ h\big). \end{aligned}$$

The idea is that during reduction of $(v\ f)$, the *output* operations in f are captured by $\lambda \mathit{output}$. Thus, $\mathit{output}(\overline{0})$ operations from $(f\ ())$ are replaced by $\mathit{output}(\overline{1})$ and vice-versa, and all other writes are skipped; so in particular $v \models_{\mathcal{F}} \psi$. This behaviour is similar to that of *effect handlers* [36]: computation $(f\ ())$ is being handled by handler h. We leave for future work the study of handlers in ECPS and of their corresponding logic.

6 Concluding Remarks

To summarize, we have studied program equivalence for a higher-order CPS language with general algebraic effects and general recursion (Sect. 2). Our main contribution is a logic $\mathcal{F}$ of program properties (Sect. 3) whose induced program equivalence coincides with contextual equivalence (Theorem 1; Sect. 4). Previous work on algebraic effects concentrated on logics that are sound for contextual equivalence, but not complete [35,38]. Moreover, $\mathcal{F}$-logical equivalence also coincides with applicative bisimilarity for our language. We exemplified our results for nondeterminism, probabilistic choice, global store and I/O. A next step would be to consider local effects (e.g. [22,33,37,39]) or normal form bisimulation (e.g. [6]).

Acknowledgements. This research was supported by an EPSRC studentship, a Balliol College Jowett scholarship, and the Royal Society. We would like to thank Niels Voorneveld for pointing out example (1), Alex Simpson and Ohad Kammar for useful discussions, and the anonymous reviewers for comments and suggestions.

References

1. Abramsky, S.: The lazy λ-calculus. In: Turner, D. (ed.) Research Topics in Functional Programming. Chapter 4, pp. 65–117. Addison Wesley, Boston (1990)
2. Abramsky, S., Jung, A.: Domain theory. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, Chap. 1, vol. 3, pp. 1–168. Oxford University Press, Oxford (1994)
3. Ahman, D., Plotkin, G.: Refinement types for algebraic effects. In: TYPES (2015)
4. Biernacki, D., Lenglet, S.: Applicative bisimulations for delimited-control operators. In: Birkedal, L. (ed.) FoSSaCS 2012. LNCS, vol. 7213, pp. 119–134. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28729-9_8
5. Cartwright, R., Curien, P., Felleisen, M.: Fully abstract semantics for observably sequential languages. Inf. Comput. **111**(2), 297–401 (1994)
6. Dal Lago, U., Gavazzo, F.: Effectful normal form bisimulation. In: Proceedings of ESOP 2019 (2019)
7. Dal Lago, U., Gavazzo, F., Levy, P.: Effectful applicative bisimilarity: monads, relators, and Howe's method. In: LICS (2017)
8. Dal Lago, U., Gavazzo, F., Tanaka, R.: Effectful applicative similarity for call-by-name lambda calculi. In: ICTCS/CILC (2017)
9. Freyd, P.: Algebraically complete categories. In: Carboni, A., Pedicchio, M.C., Rosolini, G. (eds.) Category Theory. LNM, vol. 1488, pp. 95–104. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0084215
10. Goncharov, S., Schröder, L.: A relatively complete generic Hoare logic for order-enriched effects. In: LICS (2013)
11. Gordon, A.: Operational equivalences for untyped and polymorphic object calculi. In: Gordon, A., Pitts, A. (eds.) Higher Order Operational Techniques in Semantics, pp. 9–54. Cambridge University Press, Cambridge (1998)
12. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985)
13. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
14. Howe, D.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. **124**(2), 103–112 (1996)
15. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: A logical step forward in parametric bisimulations. Technical report MPI-SWS-2014-003, January 2014
16. Hyland, M., Levy, P.B., Plotkin, G., Power, J.: Combining algebraic effects with continuations. Theoret. Comput. Sci. **375**, 20–40 (2007)
17. Johann, P., Simpson, A., Voigtländer, J.: A generic operational metatheory for algebraic effects. In: LICS (2010)
18. Lafont, Y., Reus, B., Streicher, T.: Continuations semantics or expressing implication by negation. Technical report 9321, Ludwig-Maximilians-Universität, München (1993)
19. Lassen, S.: Relational reasoning about functions and nondeterminism. Ph.D. thesis, University of Aarhus, BRICS, December 1998

20. Lassen, S.B., Levy, P.B.: Typed normal form bisimulation. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 283–297. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74915-8_23
21. Matache, C.: Program equivalence for algebraic effects via modalities. Master's thesis, University of Oxford, September 2018. https://arxiv.org/abs/1902.04645
22. Melliès, P.-A.: Local states in string diagrams. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 334–348. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_23
23. Merro, M.: On the observational theory of the CPS-calculus. Acta Inf. **47**(2), 111–132 (2010)
24. Møgelberg, R.E., Staton, S.: Linear usage of state. Log. Meth. Comput. Sci. **10** (2014)
25. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
26. Morris, J.: Lambda calculus models of programming languages. Ph.D. thesis, MIT (1969)
27. Nanevski, A., Morrisett, J., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5–6), 865–911 (2008)
28. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
29. Pitts, A.: Evaluation logic. In: Birtwistle, G. (ed.) IVth Higher Order Workshop, Banff 1990. Springer, Heidelberg (1991). https://doi.org/10.1007/978-1-4471-3182-3_11
30. Pitts, A.: Howe's method for higher-order languages. In: Sangiorgi, D., Rutten, J. (eds.) Advanced Topics in Bisimulation and Coinduction. Chapter 5, pp. 197–232. Cambridge University Press, Cambridge (2011)
31. Plotkin, G.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977)
32. Plotkin, G., Power, J.: Adequacy for algebraic effects. In: Honsell, F., Miculan, M. (eds.) FoSSaCS 2001. LNCS, vol. 2030, pp. 1–24. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45315-6_1
33. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_24
34. Plotkin, G., Power, J.: Algebraic operations and generic effects. Appl. Categ. Struct. **11**(1), 69–94 (2003)
35. Plotkin, G., Pretnar, M.: A logic for algebraic effects. In: LICS (2008)
36. Plotkin, G., Pretnar, M.: Handling Algebraic Effects. Log. Methods Comput. Sci. 9(4) (2013)
37. Power, J.: Indexed Lawvere theories for local state. In: Models, Logics and Higher-Dimensional Categories, pp. 268–282. AMS (2011)
38. Simpson, A., Voorneveld, N.: Behavioural equivalence via modalities for algebraic effects. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 300–326. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_11
39. Staton, S.: Instances of computational effects. In: Proceedings of LICS 2013 (2013)
40. Staton, S., Levy, P.B.: Universal properties for impure programming languages. In: Proceedings of POPL 2013 (2013)

41. Yachi, T., Sumii, E.: A sound and complete bisimulation for contextual equivalence in λ-calculus with call/cc. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 171–186. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-47958-3_10

42. Yoshida, N., Honda, K., Berger, M.: Logical reasoning for higher-order functions with local state. Log. Methods Comput. Sci. **4**(4) (2008)