Effects in Call-by-Push-Value*†

Robert Harper

Spring, 2024

1 Introduction

The cbpv framework (and its close relatives described in Harper (2024a)) is a general setting in which to represent effects in programming languages. The key idea is to separate types that classify values from types that classify computations. The value types define the sorts of objects that can be the results of computations and that can be bound to variables within a computation. The computation types classify effects, which act on the execution state of a program. This note is concerned with adding different sorts of effects to the language, and seeing how they are accounted for in a dynamics of the language with those constructs. Such effects are governed by equational laws, which are justified relative to the dynamics by associating behavioral invariants to types.

As mentioned in Harper (2024a) effects can be (informally) classified into two categories, *control effects* and *storage effects*. Examples of control effects are (1) general fixed point (recursion) operations that give rise to non-termination as an effect; (2) raising and handling exceptions; (3) seizing the execution state of a program as a continuation that can be activated at will. Examples of storage effects are (1) printing to "standard output"; (2) reading from "standard input"; and (3) allocating, reading and writing mutable storage cells any any value type. The control effects are all managed by augmenting the state with an explicit control stack that governs the course of a computation. The storage effects are all managed by maintaining an array of cells whose contents can be read and written during computation. In each case the dynamics takes the form of a state transition system acting on states whose form is determined by the effects under consideration.

Once the dynamics has been defined it makes sense to consider equational theories governing the effectful operations and their interactions with each other and the other constructs of the cbpv framework. These equations are intended to predict that certain computations engender the same behavior whenever they arise. The exact nature of "sameness" and "behavior" is determined by the method of logical relations, which associates execution invariants to types. For example, when control effects are considered, computations are considered equal whenever they deliver the same outcome in related stacks, and stacks are related whenever they deliver the same outcome when passed related values. When storage effects are considered, two computations have the same behavior when they have the same effects on storage cells and deliver the same answers. In all cases the main difficulty is in defining suitable relations; as effects become more complex both in themselves and in their interactions it becomes increasingly challenging to ensure that the required invariants are properly defined.

†Thanks to Yue Yao for suggesting a marked improvement to an earlier draft of this note.

2 General Dynamics for CBPV

A dynamics for the "skeletal" instance of cbpv in which there are no actual effects, and types are limited to $\mathsf{F}(A)$ and $\mathsf{U}(X)$, consists of

1. An *evaluation* relation, $M \Downarrow V$ for valuable expressions to their values. Values include $\mathsf{susp}(C)$ for some computation C, with others determined by their types.

2. A *transition system*, $C \longmapsto C'$, acting on closed computations.

Exercise 1. *Give a dynamics for the pure cbpv language described in Harper (2024a) according to the above plan.*

The next step is to define a family of logical relations interpreting value and computation types, respectively, as binary relations defining exact equality for each. The general setup is as follows:

1. For each value type, A,

   (a) Exact equality, $V \doteq V' \in A$, of closed values V and V' of type A is determined by A. In particular, $\mathsf{susp}(C) \doteq \mathsf{susp}(C') \in \mathsf{U}(X)$ iff $C \doteq C' \in X$.

   (b) Exact equality, $M \doteq M' \in A$, of closed valuables M, M' of type A, is defined to mean $M \Downarrow V$, $M' \Downarrow V'$, and $V \doteq V' \in A$.

2. For each computation type, X,

   (a) If $X = \mathsf{F}(A)$, then $C \doteq C' \in X$ iff $C \longmapsto^{*} \mathsf{ret}(M)$, $C' \longmapsto^{*} \mathsf{ret}(M')$, and $M \doteq M' \in A$.

   (b) If $X = A \rightharpoonup X$, then $C \doteq C' \in X$ iff $M \doteq M' \in A$ implies $\mathsf{ap}(C;M) \doteq \mathsf{ap}(C';M') \in X$.

Exact equality closed substitutions $\gamma, \gamma' : \Gamma$ is defined by $\gamma \doteq \gamma' \in \Gamma$ iff $\Gamma \vdash x : A$ implies $\gamma(x) \doteq \gamma'(x) \in A$. This, in turn, is used to define exact equality of open terms and computations as follows:

- $\Gamma \gg M \doteq M' \in A$ iff $\hat{\gamma}(M) \doteq \widehat{\gamma'}(M') \in A$ whenever $\gamma \doteq \gamma' \in \Gamma$.

- $\Gamma \gg C \doteq C' \in X$ iff $\hat{\gamma}(C) \doteq \hat{\gamma}(C') \in X$ whenever $\gamma \doteq \gamma' \in \Gamma$.

The reflexivity theorem states that well-typed valuables and computations are self-related at their types:

Theorem 1 (Reflexivity). *1. If $\Gamma \vdash M : A$, then $\Gamma \gg M \doteq M \in A$.*

2. If $\Gamma \vdash C : X$, then $\Gamma \gg C \doteq C \in X$.

Exercise 2. *Prove Theorem 1 by induction on typing derivations.*

The fundamental theorem states that all derivable equations are validated by the semantic interpretation.

Theorem 2 (FTLR). *1. If $\Gamma \vdash M \equiv M' : A$, then $\Gamma \gg M \doteq M' \in A$.*

2. If $\Gamma \vdash C \equiv C' : X$, then $\Gamma \gg C \doteq C' \in X$.

Exercise 3. *Prove Theorem 2 by induction on equality derivations. For the reflexive case, appeal to Theorem 1; symmetry is immediate from the definitions, which do not privilege one side over the other; transitivity requires using a reflexive instance of substitution equality.*

Exercise 4. *Formulate the relations associated to the other value and computation types given in Harper (2024a) in such a way that the fundamental theorem may be extended to include them.*

3 A Variety of Effects

In general the skeletal results may be extended to account for various effects according to the following plan:

1. Define the states $\mathcal{S}(X)$ for the execution of computations of type X.
2. Define the state transition relation on $\mathcal{S}(X)$ for the primitive operations of the free computation type that engender the effects under consideration.
3. Define the execution of the return and bind computations for the particular choice of effect. This amounts to exhibiting the monadic structure of the free computation type.
4. Define the execution of the other computations, such as function application, under consideration. These will not engender their own effects, but their execution in the effect context must be specified.

Having done this it is then necessary to define the logical relations appropriate to the given notion of effect. This means to define the relations associated to computation types as binary relations on states, following the pattern given in Section 2. Then prove the reflexivity and fundamental theorems appropriate to the chosen setting.

3.1 Reading and Printing

As an elementary example, it is natural to postulate commands to "print" and "read" a string when executed. Their statics is given by the following rules:

$$\text{PRINT-CMD}\ \frac{\Gamma \vdash M : \mathsf{string}}{\Gamma \vdash \mathsf{print}(M) : \mathsf{F}(\mathsf{unit})} \qquad \text{READ-CMD}\ \frac{}{\Gamma \vdash \mathsf{read} : \mathsf{F}(\mathsf{opt}(\mathsf{string}))}$$

The execution of these primitives is given in Figure 1 as a transition system on states of the form (I, O, C), where I and O are lists of values of string type, and C is a command of free computation type.[1]

Exercise 5. *Extend the dynamics to account for states whose computations are of function and product types. Verify the remark that the ambient states do not change other than by transitioning to a computation of free type.*

Two equations governing reading and printing are given in Figure 2. These equations call attention to the status of functions in cbpv as computations, rather than values, in that these primitives commute with abstraction and application.

[1] By analogy with Unix, one may think of I as stdin and O as stdout.

PRINT-IO-STEP

$$\frac{M \Downarrow V}{(I, O, \mathsf{print}(M)) \longmapsto (I, V{::}O, \mathsf{ret}(\langle\rangle))}$$

READ-IO-STEP-EMP

$$\frac{}{(\mathsf{nil}, O, \mathsf{read}) \longmapsto (\mathsf{nil}, O, \mathsf{ret}(\mathsf{nothing}))}$$

READ-IO-STEP-NON-EMP

$$\frac{}{(V{::}I, O, \mathsf{read}) \longmapsto (I, O, \mathsf{ret}(\mathsf{just}(V)))}$$

RET-IO

$$\frac{M \Downarrow V}{(I, O, \mathsf{ret}(M)) \longmapsto (I, O, \mathsf{ret}(V))}$$

BND-IO-RET

$$\frac{M_1 \Downarrow V_1}{(I, O, \mathsf{bnd}(\mathsf{ret}(M_1)\ ;\ x.E_2)) \longmapsto (I', O', [V_1/x]E_2)}$$

BND-IO-STEP

$$\frac{(I, O, E_1) \longmapsto (I', O', E_1')}{(I, O, \mathsf{bnd}(E_1\ ;\ x.E_2)) \longmapsto (I', O', \mathsf{bnd}(E_1'\ ;\ x.E_2))}$$

APP-IO

$$\frac{M \Downarrow V}{(I, O, \mathsf{ap}(\lambda(x.C);M)) \longmapsto (I', O', [V/x]C)}$$

APP-IO-STEP

$$\frac{(I, O, C) \longmapsto (I', O', C')}{(I, O, \mathsf{ap}(C;M)) \longmapsto (I', O', \mathsf{ap}(C';M))}$$

Figure 1: Input/Output Dynamics

PRINT-FUN

$$\frac{\Gamma \vdash M : \mathsf{string} \qquad \Gamma, x : A \vdash C : X}{\Gamma \vdash \mathsf{seq}(\mathsf{print}(M); \lambda(x.C)) \equiv \lambda(x.\,\mathsf{seq}(\mathsf{print}(M);C)) : A \rightharpoonup X}$$

PRINT-APP

$$\frac{\Gamma \vdash C : A \rightharpoonup X \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{string}}{\Gamma \vdash \mathsf{seq}(\mathsf{print}(N); \mathsf{ap}(C;M)) \equiv \mathsf{ap}(\mathsf{seq}(\mathsf{print}(N);C);M) : X}$$

READ-FUN

$$\frac{\Gamma, x : A, y : \mathsf{opt}(\mathsf{string}) \vdash C : X}{\Gamma \vdash \mathsf{bnd}(\mathsf{read}\,; y.\,\lambda(x.C)) \equiv \lambda(x.\,\mathsf{bnd}(\mathsf{read}\,; y.C)) : A \rightharpoonup X}$$

READ-APP

$$\frac{\Gamma \vdash C : A \rightharpoonup X \qquad \Gamma \vdash M : A}{\Gamma \vdash \mathsf{bnd}(\mathsf{read}\,; x.\,\mathsf{ap}(C;M)) \equiv \mathsf{ap}(\mathsf{bnd}(\mathsf{read}\,; x.C);M) : X}$$

Figure 2: IO Equations

Exact equality of computations is defined by induction on their type.

$$\begin{aligned} C \doteq C' \in \mathsf{F}(A) \text{ iff } \forall\, I, O & \\ & (I, O, C) \longmapsto^{*} (I', O', \mathsf{ret}(M)), \\ & (I, O, C') \longmapsto^{*} (I', O', \mathsf{ret}(M')), \text{ and } M \doteq M' \in A \end{aligned}$$

$$C \doteq C' \in A \rightharpoonup X \text{ iff } M \doteq M' \in A \text{ implies } \mathsf{ap}(C;M) \doteq \mathsf{ap}(C';M') \in X$$

Thus, free computations must exhibit the same I/O behavior, and return exactly equal results, and function computations must behave the same when applied to equal arguments. These relations are extended to open expressions in the usual way:

$$\Gamma \gg M \doteq M' \in A \text{ iff } \gamma \doteq \gamma' \in \Gamma \text{ implies } \hat{\gamma}(M) \doteq \widehat{\gamma'}(M') \in A$$

$$\Gamma \gg C \doteq C' \in X \text{ iff } \gamma \doteq \gamma' \in \Gamma \text{ implies } \hat{\gamma}(C) \doteq \widehat{\gamma'}(C') \in X$$

Exact equality of substitutions of values for variables is defined according to the types given by Γ.

The reflexivity and fundamental theorems are stated as in Section 2, albeit with the revised definitions of the semantic judgments.

Exercise 6. *Prove the reflexivity and fundamental theorems in the input/output setting. Be sure to check carefully the rules for read and print, as well as the remaining command constructs. Assume that the value types for strings is the diagonal relation, and that the option value type has the evident semantics.*

3.2 Exceptions and Continuations

Setting aside the important question of the type of values to associate with exceptions, their dynamics is easily formulated, treating a raise of an exception value analogously to a return. The bind command

$$\frac{\Gamma \vdash M : \mathsf{exn}}{\Gamma \vdash \mathsf{raise}(M) : X}\;\text{RAISE} \qquad \frac{\Gamma \vdash C : \mathsf{F}(A) \qquad \Gamma, x : A \vdash C_1 : X \qquad \Gamma, x : \mathsf{exn} \vdash C_2 : X}{\Gamma \vdash \mathsf{bndow}(C\,;x.C_1\,;x.C_2) : X}\;\text{BNDOW}$$

$$\frac{\Gamma, x : \mathsf{cont}(X) \vdash C : X}{\Gamma \vdash \mathsf{letcc}(x.C) : X}\;\text{LETCC} \qquad \frac{\Gamma \vdash M : \mathsf{cont}(X) \qquad \Gamma \vdash C : X}{\Gamma \vdash \mathsf{throw}(M;C) : X}\;\text{THROW}$$

Figure 3: Statics of Exceptions and Continuations

is generalized to account ordinary and exceptional returns, integrating a return point and a handler point in the same command. A stack-based dynamics is used to make it easier to integrate first-class continuations, which seize stacks as values, and reactivate them at will. Everything remains total, hence amenable to a propositions-as-types interpretation. The connection to classical logic is fascinating, as classical principles such as double-negation elimination arise as computation types, not value types. Thus, classical logic has no new notions of *proof* (values of a type), but rather a new notion of *proving* (computing a proof), and that makes all the difference.

As discussed in Harper (2016), the key step is to introduce a *stack*, or *continuation*, into the execution state. If the only purpose is to give a dynamics for exceptions, there is no reason to make the stack be something within the language itself, it is instead merely an auxiliary notion used in a particular dynamics. However, as soon as it is possible to seize the stack as a value, to be reactivated later, perhaps multiple times, then it is necessary for it to be a linguistic construct, a form of value, which is the approach taken here.

First, the statics. Assume given a value type exn that is not otherwise specified here (though see Section 3.5 below for one aspect of a full-fledged account of exception values.) Figure 3 defines the extension of the cbpv command language supporting exceptions and continuations. The elimination form for the free computation type, $\mathsf{F}(A)$, is generalized to account for both normal- and exceptional returns arising from execution of the given command of that type. When exceptions are to be propagated, rather than handled, one may write $\mathsf{bndow}(C\,;x.C_1\,;x.\,\mathsf{raise}(x))$ as $\mathsf{bnd}(C;x.C_1)$, and when returns are to be propagated rather than intercepted, one may write $\mathsf{bndow}(C\,;x.\,\mathsf{ret}(x)\,;x.C_2)$ as $\mathsf{hdl}(C;x.C_2)$.

The dynamics requires the auxiliary notion of a *stack*, or *continuation*, that makes explicit the control flow when executing a command. Such stacks, K, are either empty, $\bullet$, or a composition $K \circ x.C$ of a stack K accepting Y and a *frame* $x.C$ transforming X-returning computations into Y-returning computations. With the only negative type being $\mathsf{F}(A)$, frames are always of the form $y.\,\mathsf{bndow}(y\,;x.C_1\,;x.C_2)$, where y is not free in C_1 or C_2, abbreviated $\mathsf{bndow}(-\,;x.C_1\,;x.C_2)$, but once other computation types are admitted corresponding forms of frame are introduced.

Exercise 7. *Give a precise definition of the typing of continuations according to the informal description just given by defining the judgment $K \div \mathsf{F}(A)$ defining the well-formed stacks accepting values of type A.*

The dynamics for computations of free type is given by the transition system given in Figure 4 defined on states of the form $K \rhd C$, where $C : \mathsf{F}(A)$ and $K \div \mathsf{F}(A)$. In that setting both ret and raise transfer control to the two branches of the bndow computation. The dynamics of throw is simply a “context switch” that installs the given stack as the current one and returns the given value to it.

RET

$$\frac{M \Downarrow V}{K \circ \mathsf{bndow}(- \,;\, x.C_1 \,;\, x.C_2) \rhd \mathsf{ret}(M) \longmapsto K \rhd [V/x]C_1}$$

RAISE

$$\frac{M \Downarrow V}{K \circ \mathsf{bndow}(- \,;\, x.C_1 \,;\, x.C_2) \rhd \mathsf{raise}(M) \longmapsto K \rhd [V/x]C_2}$$

BNDOW

$$\frac{}{K \rhd \mathsf{bndow}(C \,;\, x.C_1 \,;\, x.C_2) \longmapsto K \circ \mathsf{bndow}(- \,;\, x.C_1 \,;\, x.C_2) \rhd C}$$

LETCC

$$\frac{}{K \rhd \mathsf{letcc}(x.C) \longmapsto K \rhd [\mathsf{cont}(K)/x]C}$$

THROW

$$\frac{M \Downarrow \mathsf{cont}(K')}{K \rhd \mathsf{throw}(M; C) \longmapsto K' \rhd C}$$

Figure 4: Dynamics of Exceptions and Continuations

Exercise 8. *Extend the formulation of continuations to the type $A \rightharpoonup X$ of procedures accepting a value of type A and yielding a computation of type X. This will require extending the formation and typing of stacks to permit application frames expecting a computation of function type, and corresponding rules for execution of the application and abstraction commands. Formulate, and spell out the significance of, double-negation elimination, using function types to express implications, and using suspension types to encapsulate computations as values.*

Some equational laws govern these constructs and their interactions, building on the general laws given in Harper (2024a), are given in Figure 5.[2]

Exercise 9. *State the analogues of the associative laws for the nesting of the bndow construct, generalizing those given for bnd in Harper (2024a).*

Exercise 10. *What additional equations are appropriate for the interaction between exceptions, continuations, and the constructs for function types considered in Exercise 8?*

Define *co-termination* of states, $s \downarrow s'$, to mean that execution of s and s' both terminate with the same answer. Exact equality of expressions and computations is then defined according to the following

[2] The construction used in Rule LETCC-LIFT is defined in Harper (2022).

BND-RET

$$\frac{}{\Gamma \vdash \mathsf{bndow}(\mathsf{ret}(M)\,;x.C_1\,;x.C_2) \equiv [M/x]C_1 : X}$$

BND-RAISE

$$\frac{}{\Gamma \vdash \mathsf{bndow}(\mathsf{raise}(M)\,;x.C_1\,;x.C_2) \equiv [M/x]C_2 : X}$$

LETCC-THROW

$$\frac{\Gamma \vdash C : X}{\Gamma \vdash \mathsf{letcc}(k.\,\mathsf{throw}(k;C)) \equiv C : X}$$

LETCC-DROP

$$\frac{\Gamma \vdash C : X}{\Gamma \vdash \mathsf{letcc}(k.C) \equiv C : X}$$

LETCC-FUSE

$$\frac{\Gamma, k_1 : \mathsf{cont}(X), k_2 : \mathsf{cont}(X) \vdash C : X}{\Gamma \vdash \mathsf{letcc}(k_1.\,\mathsf{letcc}(k_2.C)) \equiv \mathsf{letcc}(k.[k,k/k_1,k_2]C) : X}$$

LETCC-POP

$$\frac{\Gamma, k : \mathsf{cont}(X) \vdash C : X \qquad \Gamma, k : \mathsf{cont}(X), x : A \vdash C_1 : X \qquad \Gamma, k : \mathsf{cont}(X), x : \mathsf{exn} \vdash C_2 : X}{\Gamma \vdash \mathsf{letcc}(k.\,\mathsf{bndow}(\mathsf{throw}(k;C)\,;x.C_1\,;x.C_2)) \equiv \mathsf{letcc}(k.\,\mathsf{throw}(k;C)) : X}$$

LETCC-SEQ

$$\frac{\Gamma \vdash C : \mathsf{F}(A) \qquad \Gamma, k : \mathsf{cont}(X), x : A \vdash C_1 : X \qquad \Gamma, k : \mathsf{cont}(X), x : \mathsf{exn} \vdash C_2 : X}{\Gamma \vdash \mathsf{letcc}(k.\,\mathsf{bndow}(C\,;x.C_1\,;x.C_2)) \equiv \mathsf{bndow}(C\,;x.\,\mathsf{letcc}(k.C_1)\,;x.\,\mathsf{letcc}(k.C_2)) : X}$$

LETCC-LIFT

$$\frac{\begin{array}{c}\Gamma, k : \mathsf{cont}(\mathsf{F}(A)) \vdash C : \mathsf{F}(A) \\ \Gamma, x : A \vdash C_1 : X \qquad \Gamma, x : \mathsf{exn} \vdash C_2 : X \qquad C' \overset{\text{def}}{\equiv} \mathsf{bnd}(k' \circ x.C_1 ; k.C)\end{array}}{\Gamma \vdash \mathsf{bndow}(\mathsf{letcc}(k.C)\,;x.C_1\,;x.C_2) \equiv \mathsf{letcc}(k'.\,\mathsf{bndow}(C'\,;x.C_1\,;x.C_2)) : X}$$

Figure 5: Equational Laws Governing Exceptions and Continuations

plan:

$$\begin{gathered}
M \doteq M' \in \mathsf{ans} \text{ iff either } M, M' \Downarrow \mathsf{yes} \text{ or } M, M' \Downarrow \mathsf{no} \\
M \doteq M' \in \mathsf{cont}(X) \text{ iff } M \Downarrow \mathsf{cont}(K),\ M' \Downarrow \mathsf{cont}(K'), \text{ and } K \doteq K' \in \overline{X} \\
M \doteq M' \in \mathsf{U}(X) \text{ iff } M \Downarrow \mathsf{susp}(C),\ M' \Downarrow \mathsf{susp}(C'), \text{ and } C \doteq C' \in X
\end{gathered}$$

$$C \doteq C' \in X \text{ iff } K \doteq K' \in \overline{X} \text{ implies } K \triangleright C \downarrow K' \triangleright C'$$

$$\begin{aligned}
K \doteq K' \in \overline{\mathsf{F}(A)} \text{ iff } & M \doteq M' \in A \text{ implies } K \triangleright \mathsf{ret}(M) \downarrow K' \triangleright \mathsf{ret}(M') \text{ and} \\
& M \doteq M' \in \mathsf{exn} \text{ implies } K \triangleright \mathsf{raise}(M) \downarrow K \triangleright \mathsf{raise}(M')
\end{aligned}$$

The logical relations for closed constructs is extended as usual to the open case by considering all exactly equal substitution instances. The reflexivity theorem states that well-formed expressions and computations are self-related by logical equality.

Theorem 3 (Reflexivity)**.** *1. If $\Gamma \vdash M : A$, then $\Gamma \gg M \doteq M \in A$.*

2. If $\Gamma \vdash C : X$, then $\Gamma \gg C \doteq C' \in X$.

Exercise 11. *Prove Theorem 3.*

The fundamental theorem states that all derivable equations are semantically valid.

Theorem 4 (FTLR)**.** *1. If $\Gamma \vdash M \equiv M' : A$, then $\Gamma \gg M \doteq M' \in A$.*

2. If $\Gamma \vdash C \equiv C' : X$, then $\Gamma \gg C \doteq C' \in X$.

Exercise 12. *Prove Theorem 4 for the language as stated, then extend the proof to account for function types.*

3.3 Partiality

Partiality is introduced by permitting self-referential suspensions, which is sufficient to encode other forms of self-referential values such as recursive functions. The possibility of non-termination means that semantic equality must be weakened to allow two undefined computations to be equal, and otherwise similarly to equality in the total case. The existence of the relational intepretation is dependent on a crucial lemma stating that, roughly, any terminating computation involving a recursive suspension requires only finitely many unrollings of that suspension. A stack-based dynamics is used to facilitate the proof of this property.

A natural way to introduce partiality is to generalize the suspension type to be self-referential in that they are provided themselves as argument when forced.

$$\overset{\text{SUSP-REC}}{\dfrac{\Gamma, x : \mathsf{U}(X) \vdash C : X}{\Gamma \vdash \mathsf{susp}(x.C) : \mathsf{U}(X)}} \qquad \overset{\text{FORCE-REC}}{\dfrac{\Gamma \vdash M : \mathsf{U}(X)}{\Gamma \vdash \mathsf{force}(M) : X}}$$

Forcing a suspension unrolls the recursion by substitution the suspension itself into the suspended computation.

The dynamics is stated within the stack framework of Section 3.2.

$$\frac{M \Downarrow \mathsf{susp}(x.C)}{K \rhd \mathsf{force}(M) \longmapsto K \rhd [\mathsf{susp}(x.C)/x]C}\ \text{FORCE-SUSP}$$

The stack plays no active role in this transition; it is rather a technical device for facilitating the proof of Theorem 5.

Exercise 13. *Define a well-typed divergent computation and demonstrate that its execution diverges.*

Exercise 14. *Use recursive suspensions to define a generic recursive computation, $\mathit{fix}(x.C)$, with statics*

$$\frac{\Gamma, x : U(X) \vdash C : X}{\Gamma \vdash \mathit{fix}(x.C) : X}\ \text{FIX}$$

and dynamics $\mathit{fix}(x.C) \longmapsto [\mathit{susp}(\mathit{fix}(x.C))/x]C$. Then define $\mathit{fun}(f, x.C)$ to be $\mathit{fix}(f\,.\,\lambda(C.))$, and check that this behaves as a recursive function when applied to an argument.

A critical property of self-reference is called *compactness*, or *unwinding*, which states that only a finite iterated unrolling of a recursive suspension suffices for any given terminating computation. Note well, the order of quantification is that *given* a terminating computation involving a designated recursive suspension, *there exists* a finite unwinding of that suspension that suffices to achieve the same outcome.

To state this precisely requires a formulation of a finite approximation to a recursive suspension, written $\mathsf{susp}^{(n)}(x.C)$, where $n \geq 0$. The dynamics of truncated suspensions is given by the following rules:

$$\frac{M \Downarrow \mathsf{susp}^{(0)}(x.C)}{K \rhd \mathsf{force}(M) \longmapsto K \rhd \mathsf{force}(M)}\ \text{FORCE-SUSP-ZERO} \qquad \frac{M \Downarrow \mathsf{susp}^{(n+1)}(x.C)}{K \rhd \mathsf{force}(M) \longmapsto K \rhd [\mathsf{susp}^{(n)}(x.C)/x]C}\ \text{FORCE-SUSP-SUCC}$$

Exercise 15. *Give definitions for $\mathit{susp}^{(n)}(x.C)$ within the language in such a way that the above rules are admissible, rather than extensions to the language.* Hint: *Define the indexed suspension as an n-fold composition of non-self-referential suspensions.*

A terminating computation involving a truncated suspension will terminate with the same answer when the bound is removed.[3]

Exercise 16. *Define the* erasure *of a truncated suspension to be the suspension with the index removed, and extend it to all expressions and computations structurally. Prove for all $n \geq 0$,*

$$K \rhd [\mathit{susp}^{(n)}(x.C_0)/x]C \longmapsto^{*} \bullet \rhd \mathit{ret}(V),$$

implies

$$K \rhd [\mathit{susp}(x.C_0)/x]C \longmapsto^{*} \bullet \rhd \mathit{ret}(V).$$

[3] Following Pitts (2005) it is expedient to use the derived form of indexed suspensions in Exercise 15 in the proofs of the following lemmas.

Hint*: it will be necessary to prove the stronger formulation,*

$$[\mathit{susp}^{(n)}(x.C_0)/x]K \rhd [\mathit{susp}^{(n)}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V)$$

implies

$$[\mathit{susp}(x.C_0)/x]K \rhd [\mathit{susp}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V).$$

If a computation involving an n-bounded suspension terminates, then it will also terminate with the same answer then the bound is increased to $n + 1$.

Exercise 17. *Prove for all $n \geq 0$ if*

$$[\mathit{susp}^{(n)}(x.C_0)/x]K \rhd [\mathit{susp}^{(n)}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V)$$

then

$$[\mathit{susp}^{(n+1)}(x.C_0)/x]K \rhd [\mathit{susp}^{(n+1)}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V).$$

Compactness states that in a terminating computation only finitely many unrollings of a recursive suspension are required for the result.

Theorem 5 (Compactness)**.** *If $K \rhd [\mathit{susp}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V)$, then for some $n \geq 0$, $K \rhd [\mathit{susp}^{(n)}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V)$.*

Exercise 18. *Prove the following slightly stronger statement of compactness: if*

$$[\mathit{susp}(x.C_0)/x]K \rhd [\mathit{susp}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V),$$

then, for some $n \geq 0$,

$$[\mathit{susp}^{(n)}(x.C_0)/x]K \rhd [\mathit{susp}^{(n)}(x.C_0)/x]C \longmapsto^* \bullet \rhd \mathit{ret}(V).$$

All cases follow routinely by induction; consider only the case that $C = \mathit{force}(x)$, which is to say that the distinguished suspension is being forced. Use Exercise 17 to increase indices as necessary. See Pitts (2005) for the proof of a similar result.

There is only one equation governing self-referential suspensions,

$$\frac{\Gamma, x : \mathsf{U}(X) \vdash C : X}{\Gamma \vdash \mathsf{force}(\mathsf{susp}(x.C)) \equiv [\mathsf{susp}(x.C)/x]C : X}\;\text{FORCE-SUSP-REC}$$

The logical relations are defined according to the principles given in Section 3.2, albeit modified to account for partiality. To this end define *Kleene equivalence* between states,

$$K \rhd C \simeq K' \rhd C' \quad \text{iff} \quad K \rhd C \longmapsto^* \bullet \rhd V \;\text{ iff }\; K' \rhd C' \longmapsto^* \bullet \rhd V$$

where V is either yes or no. Using this notation, the formulation of exact equality given in Section 3.2 becomes

$$M \doteq M' \in \mathsf{U}(X) \text{ iff } \mathsf{force}(M) \doteq \mathsf{force}(M') \in X$$

$$C \doteq C' \in X \text{ iff } K \doteq K' \in \overline{X} \text{ implies } K \rhd C \simeq K' \rhd C'$$

$$K \doteq K' \in \overline{\mathsf{F}(A)} \text{ iff } M \doteq M' \in A \text{ implies } K \rhd \mathsf{ret}(M) \simeq K' \rhd \mathsf{ret}(M')$$

Exercise 19. *Extend the foregoing definitions to account for function types, $A \rightharpoonup X$, which may diverge when applied.*

The proofs of reflexivity of equality and of the fundamental theorem hinge on the following two lemmas whose proofs in turn rely on Theorem 5.

Lemma 6 (Truncated Suspensions)**.** *Suppose that $x : U(X) \gg C \doteq C' \in X$. Then, for all $n \geq 0$, $susp^{(n)}(x.C) \doteq susp^{(n)}(x.C') \in U(X)$.*

Proof. The proof is by induction on n, making use of the definitions of exact equality given above. The base case of $n = 0$ is immediate, for the indicated 0-truncated suspensions diverge whenever executed on any stack. Assume the theorem for n, and suppose that $K \doteq K' \in \overline{X}$, with the intention to show that $K \rhd \mathsf{force}(\mathsf{susp}^{(n+1)}(x.C)) \simeq K' \rhd \mathsf{force}(\mathsf{susp}^{(n+1)}(x.C'))$. First, note that $K \rhd \mathsf{force}(\mathsf{susp}^{(n+1)}(x.C)) \longmapsto K \rhd [\mathsf{susp}^{(n)}(x.C)/x]C$, and similarly for the right-hand side. But then by the assumption on C and C', the inductive hypothesis, and the definition of Kleene equivalence, the result follows immediately. □

Lemma 7 (Suspensions)**.** *If $\Gamma, x : U(X) \gg C \doteq C' \in X$, then $\Gamma \gg susp(x.C) \doteq susp(x.C') \in U(X)$.*

Proof. Suppose that $\gamma \doteq \gamma' \in \Gamma$, and define, for brevity, $\hat{C} \overset{\text{def}}{=} \hat{\gamma}(C)$ and $\widehat{C'} \overset{\text{def}}{=} \widehat{\gamma'}(C')$. Suppose further that $K \doteq K' \in \overline{X}$; it suffices to show

$$K \rhd \mathsf{force}(\mathsf{susp}(x.C)) \simeq K \rhd \mathsf{force}(\mathsf{susp}(x.C')).$$

Suppose that $K \rhd \mathsf{susp}(x.C) \longmapsto^{*} \bullet \rhd \mathsf{ret}(V)$ for some answer V. By Theorem 5 there is $n \geq 0$ such that

$$K \rhd \mathsf{force}(\mathsf{susp}^{(n)}(x.C)) \longmapsto^{*} \bullet \rhd \mathsf{ret}(V).$$

By Lemma 6 and the assumption, it follows that

$$K \rhd \mathsf{force}(\mathsf{susp}^{(n)}(x.C')) \longmapsto^{*} \bullet \rhd \mathsf{ret}(V),$$

and hence by Theorem 16

$$K \rhd \mathsf{force}(\mathsf{susp}(x.C')) \longmapsto^{*} \bullet \rhd \mathsf{ret}(V),$$

The converse is proved identically, establishing the lemma. □

It is then straightforward to formulate and prove reflexivity and the fundamental theorem for the logical relations defined above.

Exercise 20. *State the reflexivity and fundamental theorems for logical relations in this setting, and give a proof of the cases involving the function type.*

$$\text{REC-FOLD}\quad \frac{\Gamma \vdash C : [\mathsf{rec}(u.X)/u]X}{\Gamma \vdash \mathsf{fold}(C) : \mathsf{rec}(u.X)} \qquad \text{REC-UNFOLD}\quad \frac{\Gamma \vdash C : \mathsf{rec}(u.X)}{\Gamma \vdash \mathsf{unfold}(C) : [\mathsf{rec}(u.X)/u]X}$$

$$\text{UNFOLD-FOLD}\quad \frac{}{\mathsf{unfold}(\mathsf{fold}(C)) \longmapsto C} \qquad \text{UNFOLD-ARG}\quad \frac{C \longmapsto C'}{\mathsf{unfold}(C) \longmapsto \mathsf{unfold}(C')}$$

Figure 6: Recursive Computation Types

3.4 Recursive Types

The compactness theorem (Theorem 5) states that, as far as specifications of program behavior are concerned, there is nothing more to say about a recursive suspension than can be gleaned from all of its finite unrollings. This observation is critical to the definition of exact equality at suspension types, which otherwise would be circular and hence not properly defined. Similar issues arise in the definition of exact equality for general recursive types—which stands to reason in that recursive suspensions are definable in the presence of recursive types using self-application—and a similar indexed method is used to define it.

First, in a cbpv setting unrestricted recursive types arise as computation types of the form $\mathsf{rec}(u.X)$, where u is a type variable bound within X that refers to the recursive type itself. Unlike inductive and coinductive types, there is no restriction on the occurrences of u within X. Consequently, divergent computations may be defined using recursive types, and hence only make sense in a setting that embraces partiality. The introductory and eliminatory forms for recursive types are computations that fold and unfold elements of these types, and hence must be regarded as computations. The statics and dynamics of recursive types in the cbpv setting are given in Figure 6. The dynamics is defined directly on computations, rather than via a stack, because there is no need to prove compactness in this setting; rather, exact equality is defined in indexed form directly to resolve circularity.

Exercise 21. *Let $\mathit{self}(X)$ be the recursive type $\mathit{rec}(u.\ U(u) \rightharpoonup X)$. Let $\mathit{fix}(x.C)$ be $\mathit{ap}(\mathit{unfold}(\mathit{force}(S));S)$, and define $S : U(\mathit{self}(X))$ such that $\mathit{fix}(x.C) \longmapsto [\mathit{susp}(\mathit{fix}(x.C))/x]C$.*

The equational theory of recursive types expresses that the fold and unfold operations are mutally inverse:

$$\text{UNFOLD-FOLD}\quad \frac{\Gamma \vdash C : [\mathsf{rec}(u.x)/u]X}{\Gamma \vdash \mathsf{unfold}(\mathsf{fold}(C)) \equiv C : [\mathsf{rec}(u.X)/u]X} \qquad \text{FOLD-UNFOLD}\quad \frac{\Gamma \vdash C : \mathsf{rec}(u.X)}{\Gamma \vdash \mathsf{fold}(\mathsf{unfold}(C)) \equiv C : \mathsf{rec}(u.X)}$$

The question is how to justify these equations in terms of the dynamics given in Figure 6. The most obvious formulation suffers from circularity:

$$C \doteq C' \in \mathsf{rec}(u.X) \text{ iff } \mathsf{unfold}(C) \doteq \mathsf{unfold}(C') \in [\mathsf{rec}(u.X)/u]X$$

The difficulty is that the type $[\mathsf{rec}(u.X)/u]X$ is larger than $\mathsf{rec}(u.X)$ whenever u occurs within X, disrupting the usual strategy of defining these relations by induction on the structure of the type. The solution is to index exact equality of computations by $n \geq 0$, specifying a *recursion level* that is used to resolve the circularity. When $n = 0$, any two computations are deemed equal; otherwise it is defined as before for each type X and for each positive n, except that exact equality of recursive types reduces the recursion level when unfolded:

$$C \doteq C' \in_{n+1} \mathsf{rec}(u.X) \text{ iff } \mathsf{unfold}(C) \doteq \mathsf{unfold}(C') \in_n [\mathsf{rec}(u.X)/u]X$$

Exact equality of value types is similarly indexed, with suspension types handled as follows:

$$\mathsf{susp}(C) \doteq \mathsf{susp}(C') \in_n \mathsf{U}(X) \text{ iff } C \doteq C' \in_n X.$$

All other clauses remain unchanged, albeit with the recursion level playing a passive role. As ever, the indexed semantic membership judgments are defined as the indexed reflexive instances of exact equality.

Exact equality is extended to open valuables and open computations at all recursion levels.

$$\Gamma \gg M \doteq M' \in A \text{ iff } \forall n \geq 0 \text{ if } \gamma \doteq \gamma' \in_n \Gamma \text{ then } \hat{\gamma}(M) \doteq \widehat{\gamma'}(M') \in_n A$$
$$\Gamma \gg C \doteq C' \in X \text{ iff } \forall n \geq 0 \text{ if } \gamma \doteq \gamma' \in_n \Gamma \text{ then } \hat{\gamma}(C) \doteq \widehat{\gamma'}(C') \in_n X$$

That is, for all recursion levels, equal substitutions at that level give rise to equal valuables (computations) at that level.

With this in hand it is a simple matter to prove the reflexivity and fundamental theorems in the indexed form just given.

Exercise 22. *State and prove the appropriate reflexivity and fundamental theorems for recursive types.* Hint*: the inductive hypotheses for any rule states the validity of the premises for all recursion levels; this is needed to handle the indexed treatment of equality at recursive type.*

3.5 Symbol Generation

As with partiality, dynamic symbol generation is a fundamental effect that is often used to define higher-level notions of effect such as dynamically classified values or dynamically allocated mutable cells. To account for symbols in the cbpv framework, the typing judgments are indexed by a *signature*, Σ, consisting of a finite sequence of declarations $a \sim A$ associating a type, A, to the symbol, a. The associated type of a symbol a is to be uniquely determined by Σ; consequently, a signature permits at most one such association for a given symbol. The significance of the associated type depends on the situation. For example, when symbols serve as names for mutable cells, the associated type is that of the contents of the cell, and when symbols serve as classes, the associated type is that of the classified data.

Being a “generic” form of effect, new symbols are allocated by the computation $\mathsf{new}_A(a.C)$, which introduces the symbol a with associated type A for use within the computation C. As the notation suggests, the symbol a is bound within C, and may always be α-varied to ensure that it is, in fact, “new” relative to the ambient signature of symbols. The statics and dynamics of symbol generation are given in Figure 7. In this setting the judgments of the statics are indexed by the signature of active symbols, which is extended within the body of an allocation. The dynamics is given as a transition system on states of the form $\nu\, \Sigma \{ C \}$ consisting of a signature and a computation. The dynamically active symbols have *global* scope to allow values containing symbols to be used without restriction. Although not

NEW
$$\frac{\Gamma \vdash_{\Sigma,a\sim A} C : X}{\Gamma \vdash_{\Sigma} \mathsf{new}_{A}(a.C) : X}$$

OK
$$\frac{\vdash_{\Sigma} C : A}{\nu\,\Sigma\{\,C\,\}\,\mathsf{ok}}$$

INIT
$$\frac{C : \mathsf{ans}}{\nu\,\varepsilon\{\,C\,\}\,\mathsf{initial}}$$

FINAL
$$\frac{V\ \mathsf{val}_{\Sigma}}{\nu\,\Sigma\{\,\mathsf{ret}(V)\,\}\,\mathsf{final}}$$

NEW-EXEC
$$\frac{}{\nu\,\Sigma\{\,\mathsf{new}_{A}(a.C)\,\} \longmapsto \nu\,\Sigma, a \sim A\{\,C\,\}}$$

Figure 7: Symbol Generation: Statics and Dynamics

QUOTE
$$\frac{\Sigma \vdash a \sim A}{\Gamma \vdash_{\Sigma} \mathsf{quote}\langle a\rangle : \mathsf{sym}(A)}$$

GENSYM
$$\frac{}{\Gamma \vdash_{\Sigma} \mathsf{gensym}_{A} : \mathsf{sym}(A)}$$

EQ
$$\frac{\Gamma \vdash_{\Sigma} M_1 : \mathsf{sym}(A) \qquad \Gamma \vdash_{\Sigma} M_2 : \mathsf{sym}(A)}{\Gamma \vdash_{\Sigma} \mathsf{eq}(M_1\,;M_2) : \mathsf{bool}}$$

Figure 8: Symbol Type Statics

needed here, evaluation of terms is similarly indexed by a signature, written $M \Downarrow_{\Sigma} V$, to allow for values that contain symbols.

With this in hand one may consider a variety of language concepts that make use of symbols. Perhaps the most immediate application is to introduce a type, $\mathsf{sym}(A)$, whose values are *quoted symbols*, written $\mathsf{quote}\langle a\rangle$. Symbol values are introduced by gensym_{A}, which allocates a new symbol with associated type A and returns the corresponding symbol value. Two symbol values may be compared for equality with $\mathsf{eq}(M_1\,;M_2)$, which returns a boolean. The statics of these constructs is given in Figure 8, and their dynamics is given in Figure 9. Note that whereas symbol generation is a proper computation, the equality test of two valuable expressions is itself valuable, for as the dynamics makes clear no effects are involved in its evaluation.

Some equations governing symbol generation are given in Figure 10. These may be justified using Kripke-style logical relations in which the possible worlds are signatures ordered by $\Sigma' \leq \Sigma$ iff $\Sigma \vdash a \sim A$ implies $\Sigma' \vdash a \sim A$. Exact equality of values at a world, $M \doteq M' \in A\ [\Sigma]$, is defined by induction on

SYM-VAL

$$\frac{}{\mathsf{quote}\langle a\rangle\ \mathsf{val}_{\Sigma,a\sim\tau}}$$

SYM-GEN

$$\frac{}{\nu\,\Sigma\,\{\,\mathsf{gensym}_A\,\} \longmapsto \nu\,\Sigma, a \sim A\,\{\,\mathsf{ret}(\mathsf{quote}\langle a\rangle)\,\}}$$

SYM-EQ-TT

$$\frac{M_1 \Downarrow_\Sigma \mathsf{quote}\langle a\rangle \qquad M_2 \Downarrow_\Sigma \mathsf{quote}\langle a\rangle}{\mathsf{eq}(M_1\,;M_2) \Downarrow_\Sigma \mathsf{true}}$$

SYM-EQ-FF

$$\frac{M_1 \Downarrow_\Sigma \mathsf{quote}\langle a_1\rangle \qquad M_2 \Downarrow_\Sigma \mathsf{quote}\langle a_2\rangle \qquad (a_1 \neq a_2)}{\mathsf{eq}(M_1\,;M_2) \Downarrow_\Sigma \mathsf{false}}$$

Figure 9: Symbol Type Dynamics

GENSYM-NEW

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{gensym}_A \equiv \mathsf{new}_A(a.\,\mathsf{ret}(\mathsf{quote}\langle a\rangle)) : \mathsf{F}(\mathsf{sym}(A))}$$

EQ-TRUE

$$\frac{\Gamma \vdash_\Sigma M : \mathsf{sym}(A)}{\Gamma \vdash_\Sigma \mathsf{eq}(M\,;M) \equiv \mathsf{true} : \mathsf{bool}}$$

EQ-FALSE

$$\frac{(a_1 \neq a_2)}{\Gamma \vdash_\Sigma \mathsf{eq}(\mathsf{quote}\langle a_1\rangle\,;\mathsf{quote}\langle a_2\rangle) \equiv \mathsf{false} : \mathsf{bool}}$$

BND-NEW

$$\frac{\Gamma \vdash_{\Sigma,a_1\sim A_1} C_1 : \mathsf{F}(A_1) \qquad \Gamma, x : A_1 \vdash_\Sigma C_2 : X_2}{\Gamma \vdash_\Sigma \mathsf{bnd}(\mathsf{new}_{A_1}(a_1.C_1);x.C_2) \equiv \mathsf{new}_{A_1}(a_1.\,\mathsf{bnd}(C_1;x.C_2)) : X_2}$$

NEW-FUN

$$\frac{\Gamma, x : B \vdash_{\Sigma,a\sim A} C : X}{\Gamma \vdash_\Sigma \mathsf{new}_A(a.\,\lambda(x.C)) \equiv \lambda(x.\,\mathsf{new}_A(a.C)) : B \rightharpoonup X}$$

Figure 10: Equality of Symbol Expressions and Computations

the structure of A, with the following clauses being pertinent to the present situation:

$$M \doteq M' \in \mathsf{sym}(A)\ [\Sigma] \text{ iff } M \Downarrow_\Sigma \mathsf{quote}\langle a\rangle \text{ and } M' \Downarrow_\Sigma \mathsf{quote}\langle a\rangle,\ \text{where } \Sigma \vdash a \sim A$$
$$M \doteq M' \in \mathsf{U}(X)\ [\Sigma] \text{ iff } M \Downarrow_\Sigma \mathsf{susp}(C),\ M' \Downarrow_\Sigma \mathsf{susp}(C'),\ \text{and } \forall \Sigma' \leq \Sigma\ C \doteq C' \in X\ [\Sigma']$$

Note well that in the case of suspensions, the condition quantifies over all future worlds Σ' of Σ to ensure that the encapsulated computations are well-behaved whenever the suspension is forced, which may well be in a situation in which new symbols beyond those in Σ may have been generated.

Exact equality of computations relative to a world Σ is defined as follows:

$$C \doteq C' \in \mathsf{F}(A)\ [\Sigma] \text{ iff } \nu\,\Sigma\{\, C \,\} \longmapsto^{*} \nu\,\Sigma_1\{\, \mathsf{ret}(M) \,\},$$
$$\nu\,\Sigma\{\, C' \,\} \longmapsto^{*} \nu\,\Sigma_1\{\, \mathsf{ret}(M') \,\},\ \text{and } M \doteq M' \in A\ [\Sigma_1]$$
$$C \doteq C' \in A \rightharpoonup X\ [\Sigma] \text{ iff } M \doteq M' \in A\ [\Sigma] \text{ implies } \mathsf{ap}(C;M) \doteq \mathsf{ap}(C';M') \in X\ [\Sigma]$$

These may be extended to open terms by considering exactly equal substitutions for the variables declared in the given context.

Lemma 8 (Generalized Head Expansion)**.** *Suppose that $M \doteq M' \in A$ and $x : A \gg_\Sigma C \doteq C' \in X$ so that $[M/x]C \doteq [M'/x]C' \in X\ [\Sigma]$. Then $\mathsf{ap}(\lambda(x.C);M) \doteq \mathsf{ap}(\lambda(x.C');M') \in X\ [\Sigma]$.*

Sketch. Let $X = A_1 \rightharpoonup \dots A_n \rightharpoonup \mathsf{F}(B)$, and suppose that $M_i \doteq M_i' \in A_i\ [\Sigma]$ for each $1 \leq i \leq n$. Then by assumption

$$\mathsf{ap}(\dots \mathsf{ap}([M/x]C;M_1); \dots M_n) \doteq \mathsf{ap}(\dots \mathsf{ap}([M'/x]C';M_1'); \dots M_n') \in \mathsf{F}(B),$$

and hence by head expansion, the indicated term being the head redex,

$$\mathsf{ap}(\dots \mathsf{ap}(\mathsf{ap}(\lambda(x.C);M);M_1); \dots M_n) \doteq \mathsf{ap}(\dots \mathsf{ap}(\mathsf{ap}(\lambda(x.C');M');M_1'); \dots M_n') \in \mathsf{F}(B),$$

as may be seen immediately from the definition of exact equality at free types. □

Exercise 23. *State and prove the reflexivity theorem and fundamental theorem for the language with symbols using the definitions of exact equality of expressions and computations outlined above.* Hint*: make use of the generalized head expansion lemma at function types.*

Exercise 24. *Validate the equations in Figure 10 as exact equalities between computations.*

Exercise 25 (Challenging)**.** *How should exact equality be defined to validate the following two rules, stating that unused symbols can be dropped, and that the order of allocation does not matter?*

NEW-DROP
$$\frac{\Gamma \vdash_\Sigma C : X}{\Gamma \vdash_\Sigma \mathsf{new}_A(a.C) \equiv C : X}$$

NEW-SWAP
$$\frac{\Gamma \vdash_{\Sigma, a\sim A, b\sim B} C : X}{\Gamma \vdash_\Sigma \mathsf{new}_A(a.\ \mathsf{new}_B(b.C)) \equiv \mathsf{new}_B(b.\ \mathsf{new}_A(a.C)) : X}$$

Exercise 26. *What is an appropriate version of generalized head expansion in the presence of product types?*

$$\text{DCL}\quad \frac{\Gamma \vdash_{\Sigma} M : A \qquad A\ \mathsf{ground} \qquad \Gamma \vdash_{\Sigma,a\sim A} C : X}{\Gamma \vdash_{\Sigma} \mathsf{dcl}(M;a.C) : X}$$

$$\text{GET}\quad \frac{\Sigma \vdash a \sim A}{\Gamma \vdash_{\Sigma} \mathsf{get}\langle a\rangle : \mathsf{F}(A)} \qquad\qquad \text{SET}\quad \frac{\Sigma \vdash a \sim A \qquad \Gamma \vdash_{\Sigma} M : A}{\Gamma \vdash_{\Sigma} \mathsf{set}\langle a\rangle(M) : \mathsf{F}(A)}$$

Figure 11: Modernized Algol Statics (Key Rules)

$$\text{GET}\quad \frac{\Sigma \vdash a \sim A \qquad \mu(a) = V}{\{\, \mu \parallel \mathsf{get}\langle a\rangle \,\} \underset{\Sigma}{\longmapsto} \{\, \mu \parallel \mathsf{ret}(V) \,\}} \qquad \text{SET}\quad \frac{\Sigma \vdash a \sim A \qquad M \Downarrow_{\Sigma} V \qquad \mu'(a) = V,\ \mu'(b) = \mu(b)\ \text{ow}}{\{\, \mu \parallel \mathsf{set}\langle a\rangle(M) \,\} \underset{\Sigma}{\longmapsto} \{\, \mu' \parallel \mathsf{ret}(V) \,\}}$$

Figure 12: Dynamics for a Fixed Signature

3.6 Mutable State

The statics of a cbpv formulation of Modernized Algol (Harper, 2016) with free assignables is summarized in Figure 11. Typing judgments are indexed by a signature, Σ, associating ground types to assignables by a sequence of declarations $a \sim A$. A *ground* type is a value type constructed from value types other than suspension or total function types; these include finite sums and products of ground types, and inductive types constructed from other ground types. The significance of this restriction will emerge when formulating exact equality for computations that allocate and mutate memory cells.

Exercise 27. *Give an inductive definition of the judgment A ground stating that A is a ground type. Then prove that equality of values of ground types is decidable by defining a total function $\mathit{eq}_A : A \otimes A \to \mathit{bool}$ by induction on the derivation of A ground.*

The formulation of Modernized Algol will be considered in two stages: first, for a pre-allocated collection of assignables of ground type, and second, permitting allocation of such assignables with global scope.

In the first instance the dynamics is given by a signature-indexed transition relation between states of the form $\mu \parallel C$ consisting of a memory and a command that acts on it, written

$$\{\, \mu \parallel C \,\} \underset{\Sigma}{\longmapsto} \{\, \mu' \parallel C' \,\}.$$

Such states are assumed well-formed in the sense that $\vdash_{\Sigma} C : X$ for some computation type X, and μ is a composition of cells, $a_1 \hookrightarrow M_1 \parallel \cdots \parallel a_n \hookrightarrow M_n$, such that $\Sigma \vdash a_i \sim A_i$ and $\vdash_{\varepsilon} M_i : A_i$ for each $1 \leq i \leq n$.[4] The definition of the dynamics of get and set for a fixed signature is given in Figure 12.[5]

[4] The assignables in a well-formed signature are distinct from each other, so no two cells govern the same assignable.

[5] The notation $\mu(a) = M$ means that μ assigns M to assignable a.

Some illustrative equations governing the dynamics of get and set are given in Figure 13. These equations express critical properties of set and get in terms of the ambient sequentialization of the cbpv framework. Informally, these equations allow a sequence of set and get operations to be put into a simplified form consisting of a sequence of get's followed by a sequence of set's, the idea being to read the memory so as to provide the data required to modify it.

The justification of these equations is given in terms of the following formulation of exact equality for a fixed signature Σ:

$$M \doteq M' \in A \text{ iff } \dots \text{ according to } A \ \dots$$

$$\begin{aligned} C \doteq C' \in \mathsf{F}(A) \text{ iff } \mu \doteq \mu' \in \Sigma \text{ implies} \\ & \mu \parallel C \longmapsto^{*} \mu_1 \parallel \mathsf{ret}(M),\ \mu' \parallel C' \longmapsto^{*} \mu_1' \parallel \mathsf{ret}(M'), \\ & \mu_1 \equiv \mu_1' : \Sigma \text{ and } M \doteq M' \in A \end{aligned}$$

$$C \doteq C' \in A \rightharpoonup X \text{ iff } M \doteq M' \in A \text{ implies } \mathsf{ap}(C;M) \doteq \mathsf{ap}(C';M') \in X$$

$$\mu \doteq \mu' \in \Sigma \text{ iff } \Sigma \vdash a \sim A \text{ implies } \mu(a) \doteq \mu'(a) \in A$$

Two principles reflected in these definitions are that two computations, taken in isolation, are related with respect to all possible exactly equal memories, and that two computations are required to result in the same memory only once they have both completed.

Because the signature of assignables is fixed throughout, exact equality of valuables and of memories is defined with its dependency on it left implicit. In particular exact equality of suspensions is defined in the evident way:

$$M \doteq M' \in \mathsf{U}(X) \quad \text{iff} \quad M \Downarrow_{\Sigma} \mathsf{susp}(C),\ M' \Downarrow_{\Sigma} \mathsf{susp}(C'),\ C \doteq C' \in X$$

In particular the encapsulated computations are compared with respect to arbitrary memories that are exactly equal according to the signature. However, this raises an important issue with the purported definition of exact equality: it is not clear that it is well-defined! The difficulty is that the types of the memory cells are not constituent types of the classifier of the values or computations being compared, and so it is not immediately clear that the conditions given above determine a unique notion of exact equality.

One solution, adopted here, is to restrict the contents of memory cells to ground type. If we restrict memories to ground type, then $\mu \doteq \mu' \in \Sigma$ is equivalent to $\mu \equiv \mu' : \Sigma$, and the mentioned difficulties with the definition are avoided. With these points in mind, it is then possible to formulate the reflexivity and fundamental theorems for the case of a fixed collection of assignables of ground type.

Exercise 28. *Show that if suspensions were permitted to be stored in memory, then it is possible to define general recursion, and hence to define non-terminating computations. (Note, however, that valuable expressions remain terminating.)*

Exercise 29. *Prove that exactly equal valuable expressions of ground type are definitionally equivalent.*

Exercise 30. *State and prove representative cases of reflexivity and the fundamental theorem for the (revised) formulation of exact equality discussed above.* Hint*: Make use of a generalized head expansion lemma 8 suitable for this setting.*

SET-GET-SAME

$$\frac{\Sigma \vdash a \sim A}{\Gamma \vdash_{\Sigma} \mathsf{seq}(\mathsf{set}\langle a\rangle(M);\, \mathsf{get}\langle a\rangle) \equiv \mathsf{set}\langle a\rangle(M) \,:\, \mathsf{F}(A)}$$

SET-GET-DIFF

$$\frac{\Sigma \vdash a \sim A \qquad \Sigma \vdash b \sim B \qquad (a \neq b) \qquad \Gamma \vdash_{\Sigma} M \,:\, A}{\Gamma \vdash_{\Sigma} \mathsf{seq}(\mathsf{set}\langle a\rangle(M);\, \mathsf{get}\langle b\rangle) \equiv \mathsf{bnd}(\mathsf{get}\langle b\rangle; y.\, \mathsf{seq}(\mathsf{set}\langle a\rangle(M);\, \mathsf{ret}(y))) \,:\, \mathsf{F}(B)}$$

SET-SET-SAME

$$\frac{\Gamma \vdash a \sim A \qquad \Gamma \vdash_{\Sigma} M \,:\, A \qquad \Gamma, x \,:\, A \vdash N \,:\, A}{\Gamma \vdash_{\Sigma} \mathsf{bnd}(\mathsf{set}\langle a\rangle(M); x.\, \mathsf{set}\langle a\rangle(N)) \equiv \mathsf{letv}(M;\, x.\, \mathsf{set}\langle a\rangle(N)) \,:\, \mathsf{F}(A)}$$

SET-SET-DIFF

$$\frac{\Sigma \vdash a \sim A \qquad \Sigma \vdash b \sim B \qquad (a \neq b) \qquad \Gamma \vdash_{\Sigma} M \,:\, \mathsf{F}(A) \qquad \Gamma, x \,:\, A \vdash_{\Sigma} N \,:\, \mathsf{F}(B)}{\Gamma \vdash_{\Sigma} \mathsf{bnd}(\mathsf{set}\langle a\rangle(M); x.\, \mathsf{set}\langle b\rangle(N)) \equiv \mathsf{letv}(M;\, x.\, \mathsf{bnd}(\mathsf{set}\langle b\rangle(N); y.\, \mathsf{seq}(\mathsf{set}\langle a\rangle(x);\, \mathsf{ret}(y)))) \,:\, \mathsf{F}(B)}$$

GET-SET

$$\frac{\Sigma \vdash a \sim A}{\Gamma \vdash_{\Sigma} \mathsf{bnd}(\mathsf{get}\langle a\rangle; x.\, \mathsf{set}\langle a\rangle(x)) \equiv \mathsf{get}\langle a\rangle \,:\, \mathsf{F}(A)}$$

GET-GET

$$\frac{\Sigma \vdash a \sim A \qquad \Sigma \vdash b \sim B}{\Gamma \vdash_{\Sigma} \mathsf{seq}(\mathsf{get}\langle a\rangle;\, \mathsf{get}\langle b\rangle) \equiv \mathsf{get}\langle b\rangle \,:\, \mathsf{F}(A)}$$

GET-FUN

$$\frac{\Sigma \vdash a \sim A \qquad \Gamma, y \,:\, A, x \,:\, B \vdash C \,:\, X}{\Gamma \vdash_{\Sigma} \mathsf{bnd}(\mathsf{get}\langle a\rangle; y.\, \lambda(x.C)) \equiv \lambda(x.\, \mathsf{bnd}(\mathsf{get}\langle a\rangle; y.C)) \,:\, B \rightharpoonup X}$$

SET-FUN

$$\frac{\Sigma \vdash a \sim A \qquad \Gamma \vdash M \,:\, A \qquad \Gamma, y \,:\, A, x \,:\, B \vdash C \,:\, X}{\Gamma \vdash_{\Sigma} \mathsf{bnd}(\mathsf{set}\langle a\rangle(M); y.\, \lambda(x.C)) \equiv \lambda(x.\, \mathsf{bnd}(\mathsf{set}\langle a\rangle(M); y.C)) \,:\, B \rightharpoonup X}$$

Figure 13: Equations for State Operations

DCL

$$\frac{}{\nu\,\Sigma\{\,\mu \parallel \mathsf{dcl}(M;a.C)\,\} \longmapsto \nu\,\Sigma, a \sim A\{\,\mu \parallel a \hookrightarrow M \parallel C\,\}}$$

GET

$$\frac{}{\nu\,\Sigma, a \sim A\{\,\mu \parallel a \hookrightarrow M \parallel \mathsf{get}\langle a\rangle\,\} \longmapsto \nu\,\Sigma, a \sim A\{\,\mu \parallel a \hookrightarrow M \parallel \mathsf{ret}(M)\,\}}$$

SET

$$\frac{}{\nu\,\Sigma, a \sim A\{\,\mu \parallel a \hookrightarrow \_ \parallel \mathsf{set}\langle a\rangle(M)\,\} \longmapsto \nu\,\Sigma, a \sim A\{\,\mu \parallel a \hookrightarrow M \parallel \mathsf{ret}(M)\,\}}$$

Figure 14: Dynamics with Allocation

Exercise 31. *Can the foregoing be extended to account for the* total *function value type? If so, show how, and, if not, argue why it is impossible to do so.*

Exercise 32. *Extend the foregoing to account for references, &a, and their associated setref and getref operations as defined in Harper (2016). Reference types should be considered ground; check that equality of values of ground type remains decidable. Observe that reference values are simply symbols, as described in Section 3.5, albeit without, for the moment, their dynamic allocation.*

The dynamics of Modernized Algol with scope-extruding declaration of assignables is given by the transition relation between states of the form $\nu\,\Sigma\{\,\mu \parallel C\,\}$ given in Figure 14. Such states are assumed to be well-formed in the same sense as for the fixed-signature dynamics, albeit with the signature now forming part of the state. An important invariant governing the dynamics in Figure 14 is that if $\nu\,\Sigma\{\,\mu \parallel C\,\} \longmapsto \nu\,\Sigma'\{\,\mu' \parallel C\,\}$, then $\Sigma' \leq \Sigma$ in the sense that if $\Sigma \vdash a \sim A$, then $\Sigma' \vdash a \sim A$ as well (but could also associate (ground) types to assignables other than those given by Σ.)

The validity of these equations is established by defining exact equality as follows:

$$M \doteq M' \in \mathsf{U}(X)\ [\Sigma] \text{ iff } M \Downarrow_\Sigma \mathsf{susp}(C),\ M' \Downarrow_\Sigma \mathsf{susp}(C'),\ \text{and } \forall\,\Sigma' \leq \Sigma,\ C \doteq C' \in X\ [\Sigma']$$

$$\begin{aligned}
C \doteq C' \in \mathsf{F}(A)\ [\Sigma] \text{ iff } & \mu \equiv \mu' : \Sigma \text{ implies} \\
& \nu\,\Sigma\{\,\mu \parallel C\,\} \longmapsto^{*} \nu\,\Sigma_1\{\,\mu_1 \parallel \mathsf{ret}(M)\,\}, \\
& \nu\,\Sigma\{\,\mu' \parallel C'\,\} \longmapsto^{*} \nu\,\Sigma_1\{\,\mu_1' \parallel \mathsf{ret}(M')\,\}, \\
& \mu_1 \equiv \mu_1' : \Sigma_1 \text{ and } M \doteq M' \in A\ [\Sigma_1]
\end{aligned}$$

$$C \doteq C' \in A \rightharpoonup X\ [\Sigma] \text{ iff } M \doteq M' \in A\ [\Sigma] \text{ implies } \mathsf{ap}(C;M) \doteq \mathsf{ap}(C';M') \in X\ [\Sigma]$$

As in Section 3.5 exact equality of encapsulated computations is defined by quantification over future worlds to ensure that these values remain equal in any further evolution of the store engendered by the surrounding computation, as expressed by the following lemma:

DCL-RET

$$\frac{\Gamma \vdash_{\Sigma} M : A \qquad \Gamma \vdash_{\Sigma} N : A}{\Gamma \vdash_{\Sigma} \mathsf{dcl}(M;a.\,\mathsf{ret}(N)) \equiv \mathsf{ret}(N) : \mathsf{F}(A)}$$

DCL-GET

$$\frac{\Gamma \vdash_{\Sigma} M : A}{\Gamma \vdash_{\Sigma} \mathsf{dcl}(M;a.\,\mathsf{get}\langle a\rangle) \equiv \mathsf{dcl}(M;a.\,\mathsf{ret}(M)) : \mathsf{F}(A)}$$

DCL-SET

$$\frac{\Gamma \vdash_{\Sigma} M : A \qquad \Gamma \vdash_{\Sigma} N : A}{\Gamma \vdash_{\Sigma} \mathsf{dcl}(M;a.\,\mathsf{set}\langle a\rangle(N)) \equiv \mathsf{dcl}(N;a.\,\mathsf{get}\langle a\rangle) : \mathsf{F}(A)}$$

DCL-DCL

$$\frac{\Gamma \vdash_{\Sigma} M : A \qquad \Gamma \vdash_{\Sigma} N : A}{\Gamma \vdash_{\Sigma} \mathsf{dcl}(M;a.\,\mathsf{dcl}(N;b.C)) \equiv \mathsf{dcl}(N;b.\,\mathsf{dcl}(M;a.C)) : X}$$

DCL-BND

$$\frac{\Gamma \vdash_{\Sigma} M : A \qquad \Gamma \vdash_{\Sigma,a\sim A} C_1 : \mathsf{F}(A_1) \qquad \Gamma, x : A_1 \vdash_{\Sigma} C_2 : \mathsf{F}(A_2)}{\Gamma \vdash_{\Sigma} \mathsf{bnd}(\mathsf{dcl}(M;a.C_1);x.C_2) \equiv \mathsf{dcl}(M;a.\,\mathsf{bnd}(C_1;x.C_2)) : \mathsf{F}(A_2)}$$

DCL-FUN

$$\frac{\Gamma \vdash_{\Sigma} M : A \qquad \Gamma, y : B \vdash_{\Sigma,a\sim A} C : X}{\Gamma \vdash_{\Sigma} \mathsf{dcl}(M;a.\,\lambda(y.C)) \equiv \lambda(y.\,\mathsf{dcl}(M;a.C)) : B \rightharpoonup X}$$

Figure 15: Equations for Declarations

Lemma 9 (Anti-Monotonicity). *If $M \doteq M' \in A\ [\Sigma]$ and $\Sigma' \leq \Sigma$, then $M \doteq M' \in A\ [\Sigma']$.*

Exercise 33. *Prove Lemma 9. Note well the role of the quantification over future worlds in the definition of exact equality for suspensions!*

The extension of these judgments to open terms is, for the purpose of proving reflexivity, defined as follows:

$$\Gamma \gg_{\Sigma} A \in M \text{ iff } \gamma \equiv \gamma' : \Gamma\ [\Sigma],\ \text{implies } \hat{\gamma}(M) \doteq \widehat{\gamma'}(M) \in A\ [\Sigma]$$

$$\Gamma \gg_{\Sigma} X \in C \text{ iff } \gamma \equiv \gamma' : \Gamma\ [\Sigma],\ \text{implies } \hat{\gamma}(C) \doteq \widehat{\gamma'}(C) \in X\ [\Sigma]$$

Exercise 34. *Formulate and prove (representative cases of) the reflexivity theorem in the presence of declarations as well as get/set operations.*

The definition of semantic equality of open terms follows a similar pattern to the open semantic membership judgments given above.

Exercise 35. *State and prove (representative cases of) the fundamental theorem in the presence of declarations. Be sure to demonstrate the validity of the equations given in Figure 15.* Hint*: Make use of a generalized head expansion lemma 8 suitable in this setting.*

Exercise 36. *Formulate equations governing the behavior of the new, getref, and setref operations defined in Harper (2016), and prove that they are valid with respect to the extension of exact equality to account for references in the setting that also accounts for declarations.*

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Continuations, aka contradictions, aka contexts, aka stacks. Unpublished lecture note., February 2022. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/tlc-cont.pdf`.

Robert Harper. Call-by-push-value. Unpublished lecture note., January 2024a. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/cbpv.pdf`.

Robert Harper. Kripke-style logical relations for normalization. Unpublished lecture note, Spring 2024b. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/kripke.pdf`.

Paul Blain Levy. *Call-By-Push-Value*. Springer Netherlands, Dordrecht, 2003. ISBN 978-94-010-3752-5 978-94-007-0954-6. doi: 10.1007/978-94-007-0954-6. URL `http://link.springer.com/10.1007/978-94-007-0954-6`.

Benjamin C. Pierce. *Advanced topics in types and programming languages*. MIT Press, Cambridge, Mass, 2005. ISBN 978-0-262-16228-9.

A. M. Pitts. Typed Operational Reasoning. In *Advanced Topics in Types and Programming Languages*, pages 245–289. MIT Press, Cambridge, MA, 2005.

Andrew Pitts. Step-Indexed Biorthogonality: a Tutorial Example. In *Dagstuhl Seminar Proceedings (DagSemProc)*, volume 10351, pages 1–10, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. doi: 10.4230/DagSemProc.10351.6. URL `https://drops.dagstuhl.de/entities/document/10.4230/DagSemProc.10351.6`.

Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher-Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.