Assignment 5:
Continuations, Parallelism, Cost, and Modality

15-312: Principles of Programming Languages (Spring 2020)

Out: Wednesday, Nov 11, 2020
Due: Wednesday, Nov 25, 2020 11:59am ET

1 Introduction

This assignment will familiarize you with the concepts *continuations* and *parallel evaluation*. We study both concepts as extensions of **MPCF** (modal **PCF**), which we discussed in lecture before spring break. **MPCF** uses a modal separation to distinguish values and computations in a call-by-value dynamics, which is a convenient mechanism for controlling the evaluation order.

The first extension, called **KPCF**$^+$, has been discussed in recitation and introduces the explicit manipulation of continuations. It is very similar to **PCF** with continuations as discussed in lecture but uses a modal separated **K** machine. You will implement the statics and dynamics of **KPCF**$^+$. Then you will program **KPCF**$^+$ terms using `letcc` and `throw` and use your own implementation to verify correctness of your solution.

The second extension, called **MPPCF**, adds constructs for parallelism. We use a new formulation of parallelism that improves and generalizes upon the presentation in the textbook. The modal separation is used to distinguish *values*, which are data for which we have already incurred the cost of evaluation, and *expressions*, which are data for which we must still pay the cost of evaluation. You will implement the typechecker and a local dynamics for **MPPCF**, which forks tasks that evaluate sub-computations in parallel. With all of these complete, you will have implemented a parallel programming language.

Make sure to start early and to understand the statics and dynamics of the new languages. There will be plenty of code to write, so don't delay!

1.1 Submission

As usual, please submit the written part of this homework as a PDF file to Gradescope. To submit the implementation part, submit a zipfile to Gradescope. To create the zipfile, use the Makefile supplied in the handout. It will ensure that all the relevant files are handed in.

2 MPCF with Continuations

In lecture, we introduced a **K** machine to add exceptions and continuations to **PCF**. Here, we introduce a **K** machine for **MPCF**. There is no conceptual difference between continuations in **MPCF** and **PCF**. We opt for **MPCF** in this assignment since it acts as an on-ramp for **MPPCF** and since the presentation is more succinct.

2.1 Modal PCF with K Machines

Like we did for **PCF** in lecture, we first introduce a **K** machine for **MPCF**. We then add continuations in a second step. The syntax of **MPCF** is defined as follows.

τ	::=	$\texttt{nat}$	$\texttt{nat}$	naturals
	|	$\rightharpoonup(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial functions
	|	$\texttt{comp}(\tau)$	$\tau\ \texttt{comp}$	computations
v	::=	x	x	variables
	|	$\texttt{z}$	$\texttt{z}$	zero
	|	$\texttt{s}(v)$	$\texttt{s}(v)$	successor
	|	$\texttt{fun}\{\tau_1;\tau_2\}(f\,.\,x\,.\,e)$	$\texttt{fun}\,f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e$	recursive function
	|	$\texttt{comp}(e)$	$\{e\}$	suspended computation
e	::=	$\texttt{ret}(v)$	$\texttt{ret}(v)$	trivial computation
	|	$\texttt{ap}(v_1;v_2)$	$v_1(v_2)$	application
	|	$\texttt{ifz}\{e_0;x\,.\,e_1\}(v)$	$\texttt{ifz}\{\tau\}(v;e_0;x\,.\,e_1)$	zero test
	|	$\texttt{bind}(v;x\,.\,e)$	$\texttt{let}\ x \leftarrow v\ \texttt{in}\ e$	sequential evaluation
k	::=		ϵ	empty stack
	|		$k;x.e$	stack with frame
s	::=		$k \triangleright e$	Evaluating e in for k
	|		$k \triangleleft v$	Throw v to k

Evaluating an **MPCF** term using **K** Machines starts in the initial state $\epsilon \triangleright e$. The evaluation terminates when it reaches a final state specified by the dynamics after taking a number of transition steps. Several judgments are involved in describing the dynamics using **K** machines:

$\Gamma \vdash v : \tau$	Value v has type τ
$\Gamma \vdash e \mathrel{\dot{\sim}} \tau$	Expression e may evaluate to a value of τ
$k \div \tau$	Stack accepts a value of type τ
$s \longmapsto s$	Evaluation state taking a step
s ok	Evaluation state is well formed
s final	Evaluation state is final

Rules for these judgments are defined in Appendix B. Note that in the statics, $\texttt{ret}(v)$ is the only

way to elevate a value into a computation. In the dynamics, $\texttt{ret}(v)$ is the only expression that cause the state to change it's evaluation mode into throw mode.

2.2 KPCF$^+$

In this section, we define **KPCF**$^+$, a language for explicitly manipulate continuations. **KPCF**$^+$ extends **MPCF** with the following syntax:

τ	::=	$\ldots$		all **MPCF** types
	$\mid$	$\times(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	binary products
	$\mid$	$+(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	binary sums
	$\mid$	$\texttt{unit}$	$\texttt{unit}$	unit type
	$\mid$	$\texttt{void}$	$\texttt{void}$	void type
	$\mid$	$\texttt{cont}(\tau)$	$\tau\,\texttt{cont}$	continuation type
	$\mid$	$\alpha,\ \beta\ldots$		type variables
v	::=	$\ldots$		all **MPCF** values
	$\mid$	$\texttt{pair}(v_1;v_2)$	$\langle v_1, v_2\rangle$	values of binary products
	$\mid$	$\texttt{in}[\texttt{l}]\{\tau_1;\tau_2\}(v)$	$\texttt{l}\cdot v$	left injetion of sum type
	$\mid$	$\texttt{in}[\texttt{r}]\{\tau_1;\tau_2\}(v)$	$\texttt{r}\cdot v$	right injetion of sum type
	$\mid$	$\langle\rangle$	$\langle\rangle$	unit value
e	::=	$\ldots$		all **MPCF** expressions
	$\mid$	$\texttt{split}[\tau](v;x,y\,.\,e)$	$\texttt{split}\ v\ \texttt{as}\ x,y\ \texttt{in}\ e$	split a product
	$\mid$	$\texttt{case}\{x\,.\,e_1;y\,.\,e_2\}(v)$	$\texttt{case}\,v\,\{\texttt{l}\cdot x\hookrightarrow e_1 \mid \texttt{r}\cdot y\hookrightarrow e_2\}$	casing on a sum
	$\mid$	$\texttt{abort}\{\tau\}(v)$	$\texttt{case}\,v\,\{\,\}$	nullary case analysis
	$\mid$	$\texttt{letcc}\{\tau\}(x\,.\,e)$	$\texttt{letcc}\,\tau\,\texttt{in}\,x.e$	capture current continuation
	$\mid$	$\texttt{throw}\{\tau\}(v_2;v_1)$	$\texttt{throw}\,v_1\,\texttt{to}\,v_2$	throw at a continuation

There is no change to the syntax of stacks k and states s. The rules for new constructs are defined in Appendix C. In the code, we also make available regular lambda abstractions.

The types for **KPCF**$^+$ contain type variables. They allow you to write down expressions in the next subsection that do not depend on the choice of any concrete type. To account for type variables, an **KPCF**$^+$ term is type-checked under a context Δ of type variables. The typing judgments for **KPCF**$^+$ are $\Delta;\Gamma \vdash v : \tau$ and $\Delta;\Gamma \vdash e \div \tau$. A type is considered well-formed if and only if all type variables it contains appear in Δ. For the sake of simplicity, in this assignment we will fix our typing context Δ to be

$$\Delta_0 \triangleq \alpha, \beta, \gamma, \delta\ .$$

In the code, the four type variables are A,B,C, and D.

Task 2.1 (10 pts). Unlike in System T, the elimination form for products is `split` instead of left and right projection. Suppose we would like to replace `split` with left and right project v.`L` and v.`R`. Provide the statics and dynamics for those two new forms. Your rules must respect modal separation. Argue why this may not be a good idea.

In the next two tasks you will implement **KPCF**$^+$. Implementing **KPCF**$^+$ is different in a number of ways compared with implementing other languages you have encountered. In the typechecker, you may want to treat values and expressions separately due to modal separation. You not only need to be able to synthesize a type for expressions provided by the user, you will also need to verify the validity of any evaluation state (See Appendix B, Stacks and Safety). For the dynamics, your implementation will need to explicitly manage the continuation of a **KPCF**$^+$ program. You will also notice how modal separation dramatically simplifies your dynamics implementation.

Task 2.2 (20 pts). Implement the typechecker for **KPCF**$^+$ in the structure `TypeChecker` in `kpcf/language/typechecker.sml`.

Task 2.3 (20 pts). Implement the structure `Dynamics` in the file `kpcf/language/dynamics.sml`.

2.3 A Continuation of Logic

One interesting aspect of continuations is that the continuation type correspond to negation in propositional logic. In a total language, types correspond to propositions in logic, and values of a type corresponds to a proof of the corresponding proposition. This is often referred to as Curry–Howard correspondence. Specifically, if we write $A \supset B$ for A implies B, we have

$$\begin{array}{rcl} A \vee B & \text{corresponds to} & \alpha + \beta \\ A \wedge B & \text{corresponds to} & \alpha \times \beta \\ A \supset B & \text{corresponds to} & \alpha \rightharpoonup \beta \\ T & \text{corresponds to} & \texttt{unit} \\ F & \text{corresponds to} & \texttt{void} \end{array}$$

In other words, values of these regular types can be thought of as proofs of tautologies (i.e., propositions that are true under every valuation). For example, the value

$$\texttt{fun}\{\alpha;\alpha\}(\, f \,.\, x \,.\, x\,) : \alpha \to \alpha$$

is a proof of the tautology $A \supset A$.

However, this correspondence does not hold in **KPCF**$^+$ since divergence results in an inconsistent proof system. For example, the value

$$\texttt{fun}\{\alpha;\beta\}(\, f \,.\, x \,.\, f(x)\,) : \alpha \to \beta$$

would prove the proposition $A \supset B$, which is not a tautology. *Therefore, we consider a subset of* **KPCF**$^+$ *in which all functions terminate in this subsection.* **For the tasks in this subsection, you may not use recursive functions.**

If viewed as proofs, continuations correspond to *refutations* and continuation types $\texttt{cont}(\,\tau\,)$ correspond to negation of propositions. For brevity we omit uses of `bnd` and `comp` in the behavior specifications.

Task 2.4 (10 pts). Now consider the proposition $(A \supset B) \supset (B \vee \neg A)$. The *law of excluded middle*(LEM) $(A \vee \neg A)$is directly derivable from this proposition if we substitute A for B. In fact, $(A \supset B) \supset (B \vee \neg A)$ is equivalent to $A \vee \neg A$.a

In `kpcf/derivable.kpcf`, exhibit an expression e of type:

$$\cdot \vdash e \div (\alpha \rightharpoonup \beta) \rightharpoonup (\beta + \texttt{cont}(\,\alpha\,))$$

Behavior specification: Let $k \triangleright e(f)$ steps to $k \triangleleft v$. If $v = \texttt{r} \cdot k'$ for some continuation k'. Then $k' \triangleleft v'$ evaluates to $k \triangleleft \texttt{l} \cdot v''$ where v'' is the value produced by $f(v')$.

In the lecture, we have observed it's possible to "prove" the law of excluded middle (LEM) by exhibiting a term of type $\tau + \texttt{cont}(\,\tau\,)$:

$$\texttt{letcc}\{\tau + \texttt{cont}(\,\tau\,)\}(\,r\,.\,\texttt{bind}(\,\texttt{comp}(\,\texttt{letcc}\{\tau_1\}(\,r'\,.\,\texttt{throw}\,(\texttt{r} \cdot r')\,\texttt{to}\,r\,)\,);x\,.\,\texttt{l} \cdot x\,)\,)$$

Task 2.5 (10 pts). Another proposition that is equivalent to LEM is known as the Pierce's law. Although it's equivalent to LEM, it does not involve negation in the formula. In `kpcf/pierce.kpcf`, exhibit an expression e of the following type that corresponds to Pierce's law.

$$\cdot \vdash e \div ((\alpha \rightharpoonup \beta) \rightharpoonup \alpha) \rightharpoonup \alpha$$

Behavior specification: When applied to a non-diverging function f of type $(\alpha \rightharpoonup \beta) \rightharpoonup \alpha$, it returns a value of type α.
Hint: Consider how f may behave: it either returns a value of α, or it activates its argument with a value of α. Either way f knows a proof of A.

2.4 Algebraic Effects

In this section, you will implement store using continuations and exceptions in SML. Conceptually, a (integer reference) cell in a store is just a pair of functions `get : unit -> int` and `set : int -> unit -> int` such that `get` returns the value in the cell when queried and `set` updates the cell so that subsequent invocations of `get` returns the given integer (as well as additionally returning the given integer). A cell is created via `new : int -> (unit -> int) * (int -> unit -> int)`, returning the requisite services, where `get` is initialized with the given integer.

Concretely, the functionality of a cell is realized through a procedure analogous to a system call. For each service the cell provides, the kernel (us) declare an exception that is parameterized by the client data (just the client continuation for `get`, and an integer and the continuation for `set`). When invoked, the service seizes the client continuation and raises an exception coupled with the client data (context switch to the kernel), to be dealt with by a global exception handler. This handler represents the store, and is implemented as a single global (SML) reference.

Initially, the store is a reference to the empty handler:

```
val emp : exn -> int = fn _ => raise Undefined

val store = ref emp
```

Cells are added to the store by composing the associated handlers:

```
fun comp (h1, h2) : exn -> int =
  fn x => (h1 x) handle Match => (h2 x)
```

For the next two tasks, implement your solution in `algeff/algebraic.sml`.

Task 2.6 (10 pts). Implement a cell that only supports `get`, where the client data is an integer continuation.

```
(* new : int -> (unit -> int) *)
fun new n =
let
  exception Get of int cont

  (* your implementation *)
end
```

Notice that the exception `Get` is declared locally, since we want to have multiple, distinct cells in the store.

Your cell should implement the following behavior:

```
fun run c = c () handle x => (!store)(x)

val get = new 0
val get' = new 1

fun seq (c1, c2) () = (c1 () ; c2 ())

val c = seq (get, get')
(* 1 *)
val result = run c
```

Task 2.7 (10 pts). Extend your previous cell with the `set` service, where the client data is `int * int cont`.

```
(* new : int -> (unit -> int) * (int -> unit -> int) *)
fun new n =
let
  exception Get of int cont
  exception Set of int * int cont

  (*your implementation here*)
end
```

Your implementation should have the following behavior:

```
val (get, set) = new 0
val (get', set') = new 0

val c = seq (set 1, seq (set' 11, get))
```

```
(* 1 *)
val result = run c
```

3 Parallel PCF

The textbook presentation of System **PCF** gives us the means of executing computations in parallel. We'd like to generalize the binary fork-join model we've seen so far, as well as to examine the cost of parallel programs more closely. So far in **PCF**, the notion of a value was an expression that cannot be evaluated further. This is still the case in the parallel formulation, but we should now be more careful on the issue of cost.

Suppose the user provides as input a long list of number literals. Any reasonable representation of lists will represent this as a value, i.e. not provide a means of evaluating it further. However, the value judgment is defined inductively, meaning an interpreter must still examine the entire list and every element within it in order to conclude that it is a value. This process incurs linear cost. Now imagine that this list is used as an argument to a function, and substituted in several locations throughout the function body. Now the interpreter will repeatedly evaluate this object, and the cost becomes nontrivial. The notion of value, as specified by the dynamics, has diverged from a more desirable definition of value under parallelism: *a data element for which we need incur no additional cost.*

The solution is to utilize a *modally separate* semantics, as we will see in Modernized Algol. In Algol, we separate commands from expressions in that commands have effects while expressions are pure. Here, we make the claim that expressions are not values, but through the incurring of cost may eventually compute a value. That's a powerful concept, which lets us precisely state that costs should be attributed to expressions when they perform computation to result in values.

This language with the modal separation is the *by-value* interpretation of **PCF**, extended with types

that enable parallel computation. The syntax of the new language **MPPCF** is:

τ	::=	`nat`	`nat`	naturals
	|	$\rightharpoonup(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial functions
	|	`eprod`$[n](\tau_1;\ldots;\tau_n)$	$\tau_1 \otimes \ldots \otimes \tau_n$	eager products
	|	`lprod`$[n](\tau_1;\ldots;\tau_n)$	$\{\tau_1 \,\&\, \ldots \,\&\, \tau_n\}$	lazy products
	|	`seq`(τ)	τ `seq`	sequences
	|	`gen`(τ)	τ `gen`	generators
v	::=	x	x	variables
	|	`num`$[n]$	n	numeric literals
	|	`fun`$\{\tau_1;\tau_2\}(f\,.\,x\,.\,e)$	`fun` $f(x{:}\tau_1){:}\tau_2$ `is` e	recursive function[1]
	|	`etup`$[n](v_1;\ldots;v_n)$	$v_1 \otimes \ldots \otimes v_n$ *	eager tuples
	|	`ltup`$[n](e_1;\ldots;e_n)$	$\{e_1 \,\&\, \ldots \,\&\, e_n\}$	lazy tuples
	|	`seq`$\{\tau\}[n](v_1,\ldots,v_n)$	$\langle v_1,\ldots,v_n\rangle^*$	sequences
	|	`gen`$\{\tau\}(v;i\,.\,e)$	`gen`$\{\tau\}[v]$ `with` i `in` e	generators
e	::=	`ret`(v)	`ret`(v)	return value
	|	`ap`$(v_1;v_2)$	$v_1(v_2)$	application
	|	`s`(v)	`s`(v)	successor
	|	`ifz`$\{e_0;x\,.\,e_1\}(v)$	`ifz`$\{\tau\}(v;e_0;x\,.\,e_1)$	zero test
	|	`split`$[n](v;x_1,\ldots,x_n\,.\,e)$	`split` v `as` $x_1,\ldots,x_n$ `in` e	tuple unpack
	|	`len`(v)	$\lvert v\rvert$	sequence length
	|	`sub`$(v_1;v_2)$	$v_1[v_2]$	sequence subscript
	|	`parbnd`$(v;x\,.\,e)$	`parbnd` $x \leftarrow v$ `in` e	parallel evaluation
	|	`seqbnd`$(v;x\,.\,e)$	`seqbnd` $x \leftarrow v$ `in` e	parallel sequence tabulation

This language has **PCF**'s natural and function types, along with two new product types, one eager and one lazy. The distinction between the two shall be made clear shortly. We have also distinguished the value and expression sorts; they are now syntactically separate and there is no question as to where a value is expected. Values may be lifted into expressions using the *return* construct. Expressions are modally separate. Zero or more expressions may be suspended as values using lazy products.

Suspended computations are eliminated through the *bind* constructs. In **MPPCF**, there are two such constructs, having similar concrete syntax and capable of eliminating different kinds of expressions. The parallel evaluation construct, `parbnd`, eliminates a lazy tuple in favor of an eager one, computing each element in parallel. The parallel sequence tabulation construct, `seqbnd`, eliminates a sequence generator in favor of an eager sequence, also in parallel.

Natural numbers are eliminated via a zero test, eager tuples are eliminated via a pattern-matching `split` operator, and sequences may be examined for their length and elements retrieved by index.

Lazy products should be considered as a generalization of computation type in modal **PCF** to n-ary. `parbnd` on the other hand, may be considered as generalized sequential `bind` to n-ary. It might

[1] In the code, we also make available regular lambda functions, not described here for brevity.

worth noting that although those generalizations are motivated from a type-theoretic point of view, they nicely reflect parallelism from a computing perspective.

Another important fact about eager tuples and sequences is that they are *internal forms* that cannot be constructed directly. They can still be manipulated, but they will only show up as variables. Their introduction forms are the elimination forms of lazy tuples and generators.

3.1 Statics

The static semantics of many of the constructs in this language should be familiar from your previous experience with **PCF**. However, we now define the statics separately for values and for expressions, much as we do for expressions and commands in Algol.

3.1.1 Values

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{num}[\,n\,] : \mathtt{nat}} \qquad \frac{\Gamma, f : \tau_1 \rightharpoonup \tau_2, x : \tau_1 \vdash e \div \tau_2}{\Gamma \vdash \mathtt{fun}\{\tau_1; \tau_2\}(\,f\,.\,x\,.\,e\,) : \tau_1 \rightharpoonup \tau_2}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \ldots \quad \Gamma \vdash v_n : \tau_n}{\Gamma \vdash \mathtt{etup}[n](\,v_1 \otimes \ldots \otimes v_n\,) : \tau_1 \otimes \ldots \otimes \tau_n} \qquad \frac{\Gamma \vdash e_1 \div \tau_1 \quad \ldots \quad \Gamma \vdash e_n \div \tau_n}{\Gamma \vdash \mathtt{ltup}[n](\,e_1 \,\&\, \ldots \,\&\, e_n\,) : \{\tau_1 \,\&\, \ldots \,\&\, \tau_n\}}$$

$$\frac{\Gamma \vdash v_1 : \tau \quad \ldots \quad \Gamma \vdash v_n : \tau}{\Gamma \vdash \mathtt{seq}\{\tau\}[n](v_1, \ldots, v_n) : \tau\ \mathtt{seq}} \qquad \frac{\Gamma \vdash v : \mathtt{nat} \quad \Gamma, i : \mathtt{nat} \vdash e \div \tau}{\Gamma \vdash \mathtt{gen}\{\tau\}(\,v; i\,.\,e\,) : \tau\ \mathtt{gen}}$$

The value typing judgment is woven into the expression typing judgment. Here, the most interesting thing is that lazy tuples have lazy product type and eager tuples have eager product type. Likewise, sequences have sequence type and generators have generator type.

3.1.2 Expressions

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathtt{ret}(\,v\,) \div \tau} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \rightharpoonup \tau_2 \quad \Gamma \vdash v_2 : \tau_1}{\Gamma \vdash \mathtt{ap}(\,v_1; v_2\,) \div \tau_2} \qquad \frac{\Gamma \vdash v : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(\,v\,) \div \mathtt{nat}}$$

$$\frac{\Gamma \vdash v : \mathtt{nat} \quad \Gamma \vdash e_1 \div \tau \quad \Gamma, x : \mathtt{nat} \vdash e_2 \div \tau}{\Gamma \vdash \mathtt{ifz}\{e_1; x\,.\,e_2\}(\,v\,) \div \tau} \qquad \frac{\Gamma \vdash v : \tau_1 \otimes \ldots \otimes \tau_n \quad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \div \tau}{\Gamma \vdash \mathtt{split}[n](\,v; x_1, \ldots, x_n\,.\,e\,) \div \tau}$$

$$\frac{\Gamma \vdash v : \tau\ \mathtt{seq}}{\Gamma \vdash \mathtt{len}(\,v\,) \div \mathtt{nat}} \qquad \frac{\Gamma \vdash v_1 : \tau\ \mathtt{seq} \quad \Gamma \vdash v_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{sub}(\,v_1; v_2\,) \div \tau}$$

$$\frac{\Gamma \vdash v : \{\tau_1 \,\&\, \ldots \,\&\, \tau_n\} \quad \Gamma, x : \tau_1 \otimes \ldots \otimes \tau_n \vdash e \div \tau}{\Gamma \vdash \mathtt{parbnd}(\,v; x\,.\,e\,) \div \tau} \qquad \frac{\Gamma \vdash v : \tau_1\ \mathtt{gen} \quad \Gamma, x : \tau_1\ \mathtt{seq} \vdash e \div \tau}{\Gamma \vdash \mathtt{seqbnd}(\,v; x\,.\,e\,) \div \tau}$$

Returns, applications, zero test, and sequence length and subscript all behave as expected. The tuple split construct pattern matches an eager tuple against as many variables as there are elements in the tuple. Finally, the parallel evaluation construct is simultaneously an elimination for the lazy tuple and an introduction for the eager tuple. Likewise for the parallel sequence evaluation, which eliminates generators and introduces sequences.

Why do we call the lazy tuple lazy? Values of lazy product type hold n expressions (computations) suspended, able to be forced later. When we are ready to evaluate each element of the tuple in parallel, we may force its evaluation using the `parbnd` construct. Each independent element is evaluated, and an eager tuple is made available containing the results of the computations.

Note that this means the singleton lazy product, $\texttt{lprod}[1](\tau)$, acts the same way as the type of suspensions of type τ, and the singleton lazy tuple is exactly a lazy suspension. To force the computation, we must evaluate the lazy tuple, then project out its first component. Of course, since singleton tuples are so common, we provide this special case within the syntax directly.

We can express computations using tuples with sequences as well, presenting the difference between static parallelism and dynamic parallelism. Though dynamic parallelism with sequences is more flexible, it is slightly deficient in its safety guarantees and can potentially be more difficult to schedule at runtime.

3.2 Evaluation Dynamics

Though we will not implement **MPPCF** using evaluation dynamics, it is a very useful tool for understanding how the language evaluates, and which components are evaluated in parallel.

The evaluation dynamics are defined in terms of the judgment $e \Downarrow^c v$, where the cost c is the abstract cost of the operation. We use $\oplus$ to denote sequential composition and $\otimes$ to denote parallel composition.

$$\frac{}{\texttt{ret}(\,v\,) \Downarrow^{\mathbf{1}} v} \qquad \frac{[\texttt{fun}\{\tau_1;\tau_2\}(\,f\,.\,x\,.\,e\,), v_2/f, x]e \Downarrow^c v}{\texttt{ap}(\,\texttt{fun}\{\tau_1;\tau_2\}(\,f\,.\,x\,.\,e\,); v_2\,) \Downarrow^{c \oplus \mathbf{1}} v} \qquad \frac{}{\texttt{s}(\,\texttt{num}[\,n\,]\,) \Downarrow^{\mathbf{1}} \texttt{num}[\,n+1\,]}$$

$$\frac{e_1 \Downarrow^c v}{\texttt{ifz}\{e_1; x\,.\,e_2\}(\,\texttt{num}[\,0\,]\,) \Downarrow^{c \oplus \mathbf{1}} v} \qquad \frac{n \neq 0 \quad [\texttt{num}[\,n-1\,]/x]e_2 \Downarrow^c v}{\texttt{ifz}\{e_1; x\,.\,e_2\}(\,\texttt{num}[\,n\,]\,) \Downarrow^{c \oplus \mathbf{1}} v}$$

$$\frac{[v_1, \ldots, v_n/x_1, \ldots, x_n]e \Downarrow^c v}{\texttt{split}[n](\,v_1 \otimes \ldots \otimes v_n; x_1, \ldots, x_n\,.\,e\,) \Downarrow^{c \oplus \mathbf{1}} v} \qquad \frac{}{\texttt{len}(\,\langle v_1, \ldots, v_n \rangle\,) \Downarrow^{\mathbf{1}} \texttt{num}[\,n\,]}$$

$$\frac{i < n}{\langle v_0, \ldots, v_{n-1} \rangle\,[\texttt{num}[\,i\,]] \Downarrow^{\mathbf{1}} v_i} \qquad \frac{e_1 \Downarrow^{c_1} v_1 \quad \ldots \quad e_n \Downarrow^{c_n} v_n \quad [v_1 \otimes \ldots \otimes v_n/x]e \Downarrow^c v}{\texttt{parbnd}(\,\{e_1 \,\&\, \ldots \,\&\, e_n\}; x\,.\,e\,) \Downarrow^{(c_1 \otimes \ldots \otimes c_n) \oplus c \oplus \mathbf{1}} v}$$

$$\frac{[\texttt{num}[\,0\,]/i]e \Downarrow^{c_1} v_1 \quad \ldots \quad [\texttt{num}[\,n-1\,]/i]e \Downarrow^{c_n} v_n \quad [\langle v_1, \ldots, v_n \rangle/x]e' \Downarrow^c v \quad n > 0}{\texttt{seqbnd}(\,\texttt{gen}\{\tau\}[\texttt{num}[\,n\,]]\ \texttt{with}\ i\ \texttt{in}\ e; x\,.\,e'\,) \Downarrow^{(c_1 \otimes \ldots \otimes c_n) \oplus c \oplus \mathbf{1}} v}$$

Evaluation semantics precisely describe "what" the expressions should do, but not "how" to do so in parallel. For that, we use a more involved structural dynamics. As you saw, it is possible to relate the evaluation semantics to the structural semantics by proving their equivalence regarding cost assignments, but here we will take their conceptual equivalence for granted.

3.2.1 Local Transitions

The structural dynamics of **MPPCF** involve two types of transitions: local transitions, which represent the work that one processor may perform in one step on an expression, and global

transitions, which represent the work of multiple processors. The mechanism we use to study these semantics is the **P** machine.

In the **P** machine, we introduce a new expression-like notation for join-points in the computation.

$$\texttt{join}[\ldots](x.e)$$

denotes a blocked computation that depends on the result of specific computations. This is a general mechanism to carry out a subcomputation.

The introduction for a `join` is the parallel evaluation construct:

$$\left\{\begin{array}{c} \nu a\{a \hookrightarrow \texttt{parbnd}(\,\{e_1 \,\&\, \ldots \,\&\, e_n\}; x\,.\,e\,)\} \\ \longmapsto_{loc} \\ \nu a\ a_1\ \ldots\ a_n\{a \hookrightarrow \texttt{join}[a_1 \otimes \ldots \otimes a_n](x.e) \otimes a_1 \hookrightarrow e_1 \otimes \ldots \otimes a_n \hookrightarrow e_n\} \end{array}\right\}$$

In **MPPCF**, there is no explicit form for sequential composition. This is because a hypothetical sequential composition construct can be considered a special case of parallel evaluation with only one operand:

$$\left\{\begin{array}{c} \nu a\{a \hookrightarrow \texttt{bind}(\,\{e\}; x\,.\,e'\,)\} \\ \longmapsto_{loc} \\ \nu a\ a_1\{a \hookrightarrow \texttt{join}[a_1](x.e') \otimes a_1 \hookrightarrow e\} \end{array}\right\}$$

And generators evaluate their bodies a specified number of times, substituting the index each time:

$$\left\{\begin{array}{c} \nu a\{a \hookrightarrow \texttt{seqbnd}(\,\texttt{gen}\{\tau\}[\texttt{num}[\,n\,]]\ \texttt{with}\ i\ \texttt{in}\ e; x\,.\,e'\,)\} \\ \longmapsto_{loc} \\ \nu a\ a_1\ \ldots\ a_n\{a \hookrightarrow \texttt{join}[\langle a_1, \ldots, a_n\rangle](x.e') \otimes a_1 \hookrightarrow [\texttt{num}[\,0\,]/i]e \otimes \ldots \otimes a_n \hookrightarrow [\texttt{num}[\,n-1\,]/i]e\} \end{array}\right\}$$

The elimination for a `join` is when all of its operands have evaluated:

$$\left\{\begin{array}{c} \nu a\ a_1\ \ldots\ a_n\{a \hookrightarrow \texttt{join}[a_1 \otimes \ldots \otimes a_n](x.e) \otimes a_1 \hookrightarrow \texttt{ret}(\,v_1\,) \otimes \ldots \otimes a_n \hookrightarrow \texttt{ret}(\,v_n\,)\} \\ \longmapsto_{loc} \\ \nu a\{a \hookrightarrow [v_1 \otimes \ldots \otimes v_n/x]e\} \end{array}\right\}$$

$$\left\{\begin{array}{c} \nu a\ a_1\{a \hookrightarrow \texttt{join}[a_1](x.e) \otimes a_1 \hookrightarrow \texttt{ret}(\,v\,)\} \\ \longmapsto_{loc} \\ \nu a\{a \hookrightarrow [v/x]e\} \end{array}\right\}$$

$$\left\{\begin{array}{c} \nu a\ a_1\ \ldots\ a_n\{a \hookrightarrow \texttt{join}[\langle a_1, \ldots, a_n\rangle](x.e) \otimes a_1 \hookrightarrow \texttt{ret}(\,v_1\,) \otimes \ldots \otimes a_n \hookrightarrow \texttt{ret}(\,v_n\,)\} \\ \longmapsto_{loc} \\ \nu a\{a \hookrightarrow [\langle v_1, \ldots, v_n\rangle\,/x]e\} \end{array}\right\}$$

There is no local dynamics rule for `ret`; it represents an expression that has been evaluated to its fullest point. `ret`s are eliminated by the join points, and a total program should eventually evaluate to $\texttt{ret}(\, v \,)$ for some value v. Note that this definition is slightly different from the evaluation semantics, which take expressions to values, but is the same in spirit.

Each other rule merely steps some expression that can evaluate:

$$\overline{\nu a\{a \hookrightarrow \texttt{ap}(\, \texttt{fun}\{\tau_1; \tau_2\}(\, f \,.\, x \,.\, e \,); v \,)\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow [\texttt{fun}\{\tau_1; \tau_2\}(\, f \,.\, x \,.\, e \,), v/f, x]e\}}$$

$$\overline{\nu a\{a \hookrightarrow \texttt{s}(\, \texttt{num}[\, n \,]\,)\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow \texttt{ret}(\, \texttt{num}[\, n+1 \,]\,)\}}$$

$$\overline{\nu a\{a \hookrightarrow \texttt{ifz}\{e_0; x \,.\, e_1\}(\, \texttt{num}[\, 0 \,]\,)\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow e_0\}}$$

$$\overline{\nu a\{a \hookrightarrow \texttt{ifz}\{e_0; x \,.\, e_1\}(\, \texttt{num}[\, n+1 \,]\,)\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow [\texttt{num}[\, n \,]/x]e_1\}}$$

$$\overline{\nu a\{a \hookrightarrow \texttt{split}[n](\, v_1 \otimes \ldots \otimes v_n; x_1, \ldots, x_n \,.\, e \,)\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow [v_1, \ldots, v_n/x_1, \ldots, x_n]e\}}$$

$$\overline{\nu a\{a \hookrightarrow \texttt{len}(\, \langle v_1, \ldots, v_n \rangle\,)\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow \texttt{ret}(\, \texttt{num}[\, n \,]\,)\}}$$

$$\frac{i < n}{\nu a\{a \hookrightarrow \langle v_0, \ldots, v_{n-1} \rangle\, [\texttt{num}[\, i \,]]\} \underset{loc}{\longmapsto} \nu a\{a \hookrightarrow \texttt{ret}(\, v_i \,)\}}$$

3.2.2 Errors

There is one kind of error that can occur, when an illegal subscript is taken:

$$\frac{i \geq n}{\nu a\{a \hookrightarrow \langle v_0, \ldots, v_{n-1} \rangle\, [\texttt{num}[\, i \,]]\}\ \mathsf{err}}$$

Errors are propagated at join-points, and if multiple errors should result, the leftmost error should be propagated.

$$\frac{\nu \Sigma_1\{\mu_1\} \underset{loc}{\longmapsto} \nu \Sigma_1\{\mu_1'\} \quad \nu a_1\{a_1 \hookrightarrow e_1\}\ \mathsf{err}}{\nu \Sigma_1\ a_1\ \Sigma_2\{\mu_1 \otimes a_1 \hookrightarrow e_1 \otimes \mu_2\}\ \mathsf{err}}$$

3.2.3 Global Transitions

Each global transition represents the simultaneous execution of one local step of computation on each of up to $p \geq 1$ processors:

$$\frac{\begin{array}{c}\nu\Sigma_1\, a_1\{a_1 \hookrightarrow e_1 \otimes \mu_1\} \underset{loc}{\longmapsto} \nu\Sigma_1'\, a_1\{a_1 \hookrightarrow e_1' \otimes \mu_1'\} \\ \cdots \\ \nu\Sigma_n\, a_n\{a_n \hookrightarrow e_n \otimes \mu_n\} \underset{loc}{\longmapsto} \nu\Sigma_n'\, a_n\{a_n \hookrightarrow e_n' \otimes \mu_n'\}\end{array}}{\left\{\begin{array}{c}\nu\Sigma_1\, a_1\, \ldots\, \Sigma_n\, a_n\, \Sigma\{a_1 \hookrightarrow e_1 \otimes \mu_1 \otimes \ldots \otimes a_n \hookrightarrow e_n \otimes \mu_n \otimes \mu\} \\ \underset{glo}{\longmapsto} \\ \nu\Sigma_1'\, a_1\, \ldots\, \Sigma_n'\, a_n\, \Sigma\{a_1 \hookrightarrow e_1' \otimes \mu_1' \otimes \ldots \otimes a_n \hookrightarrow e_n' \otimes \mu_n' \otimes \mu\}\end{array}\right\}}$$

The rule picks the first $n \leq p$ tasks (a non-empty collection of computations in the work list) for which a local step can be taken and executes them in parallel (that is, in one global step). This represents one particular scheduling. If you allow rearrangements of the tasks, then other schedules can be represented this way as well. This rule, therefore, introduces non-determinism into the **MPPCF** dynamics.

4 Implementation

4.1 Typechecker

Task 4.1 (25 pts). Implement the typechecker for values and expressions for this language in the structure `TypeChecker` in `language/typechecker.sml`.

4.2 Local Dynamics

Your task in this section is to implement a dynamics for taking local steps. Its signature is:

```
signature LOCALSTEPPER =
sig
  exception Malformed of string

  datatype result =
    Fork of Mppcf.Exp.t list * (Mppcf.Value.t list -> Mppcf.Exp.t)
  | Continue of Mppcf.Exp.t
  | Final of Mppcf.Value.t
  | Error of exn

  (* Return when a subscript is out of range *)
  exception Subscript
  (* Return when a generator is of length zero *)
  exception Length

  val step : Mppcf.Exp.t -> result
end
```

The job of the `step` function is to implement the relevant portion of the dynamics and turn an expression into its corresponding `result`. In the `result` datatype we have four possibilities:

1. `Fork (es, join)` represents a fork in computation. `es` is a list of expressions which need to be turned into values in order to compute the function `join`. This will occur whenever there are multiple unevaluated subexpressions.
2. `Continue e` indicates that we were able to make forward progress in computation, without needing to wait for additional computations.
3. `Final v` occurs when we reach a value, indicating that we are either at the top level of computation and finished, or this term is ready for feeding to a `join` function. This decision will be made by the scheduler.
4. `Error e` occurs when an error is encountered. In practice `e` will be either `Subscript` or `Length` for the two error cases.

Notice that `step` does not need to be recursive (none of the local stepping rules have premises containing local steps!)

Also note that you should not `raise` exceptions in your local stepper (except `Malformed`). You should instead return `Error e` so that the exception can be propagated correctly by the join.

Task 4.2 (25 pts). Implement the structure `LocalStepper` in the file `language/localstepper.sml` according to the specification given above.

4.3 Derived Forms

Writing code in **MPPCF** may be inconvenient due to the modal separation. The separation between values and expressions makes even writing arithmetic computations somewhat elaborate. We'd really like some language extensions that make our lives easier. This set of derived forms encapsulate commonly repeated elements. They are:

$$
\begin{array}{rcll}
e & \texttt{+=} & e_1 + e_2 & \text{addition} \\
 & | & e_1 - e_2 & \text{subtraction} \\
 & | & e_1 \times e_2 & \text{multiplication} \\
 & | & e_1 \texttt{ / } e_2 & \text{division} \\
 & | & e_1 \texttt{ <= } e_2 & \text{comparison}
\end{array}
$$

All of these are considered expressions, not values. These constructs are completely representable in vanilla **MPPCF**, but it would be painful to write many programs without having them. Therefore, the parser accepts these constructs, and produces an ABT for the "derived" language. We then translate this language to **MPPCF**.

We do not provide the entire semantics here, but as a rule of thumb, the constructs work as they do in ML. Division by zero has undefined behavior, but otherwise division should round exactly as the `div` operator does in Standard ML (down). You are encouraged to look back at your **PCF** code from Assignment 3, but do note the difference in the division specification! Finally, e_1 `<=` e_2 should evaluate to 1 if $e_1 \leq e_2$ or 0 otherwise.

Also, the arithmetic operations are required to evaluate their arguments in parallel. We will also be doing some parallel programming in the next section.

Task 4.3 (30 pts). Implement the desugaring transformation, which converts the above expressions to basic **MPPCF**. These are the functions in the structure `DesugarOps` in `desugar/desugar-ops.sml`. See the signature in `desugar/desugar-ops.sig` and the use of these functions in `desugar/desugar.sml` to see what they do.

Hint: Make sure to read the notes in the code carefully, including helpful helper functions.

Congrats! Once the translation has taken place (or even before, if you refrain from using the derived form constructs), you will be able to write and run code in **MPPCF**.

4.4 Parallel Programming

Now that we have a fully functional parallel programming language, we can write some useful programs in it and have them run in a parallel fashion. You are provided with a parallel runtime for **MPPCF** which schedules your programs. Though for practical reasons we may not see much speedup in an interpreted language like this one, it is inspired by data-parallel languages like NESL which allow the programmer to easily write parallel programs.

You can see the scheduler which is implemented for you in the `execute/` directory. It uses the interface of processors, which are essentially threads in Concurrent ML that receive tasks and emit results, and a scheduler interface which manipulates a work list according to a k-DFS scheduling strategy. The `Run` interface then exposes the ability to evaluate an expression to a value, which is

derived from a terminal `ret` expression. Take a look at `execute/scheduler.sml` to see how the scheduler algorithm is implemented.

4.4.1 Map and Reduce

We will now do some parallel programming in **MPPCF**, taking advantage of its sequence support to build up some features of a sequence library.

Consider an n-sequence of natural numbers $\langle v_1, \ldots, v_n \rangle$ and a function $f : \texttt{nat} \rightharpoonup \texttt{nat}$. The expected behaviour of `map` is as follows:

$$\texttt{map}\ f\ \langle v_1, \ldots, v_n \rangle = \langle f(v_1), \ldots, f(v_n) \rangle$$

Theoretically we may compute all the evaluations of f in parallel, resulting in linear work and constant span.

Task 4.4 (10 pts). Write this parallel `map` function in `map.mppcf`. It should have the type $(\texttt{nat} \rightharpoonup \texttt{nat}) \rightharpoonup \texttt{nat seq} \rightharpoonup \texttt{nat seq}$ and satisfy the given work and span bounds.

Consider an n-sequence of natural numbers $\langle v_1, \ldots, v_n \rangle$ and an associative binary operator on `nat` called $*$. Assume n is a power of 2. The expected behaviour of `reduce` is as follows:

$$\texttt{reduce}\ (*)\ \langle v_1, \ldots, v_n \rangle = v_1 * \ldots * v_n$$

Since $*$ is associative, we can parenthesize this any way we want. In particular, we can parenthesize this as a balanced binary tree:

$$((v_1 * v_2) * (v_3 * v_4)) * \ldots$$

Then for each level of this tree, we can compute all the nodes at that level in parallel, for linear work and logarithmic span.

Task 4.5 (20 pts). Write this parallel `reduce` function in `reduce.mppcf`. It should have the type $(\texttt{nat} \rightharpoonup \texttt{nat} \rightharpoonup \texttt{nat}) \rightharpoonup \texttt{nat seq} \rightharpoonup \texttt{nat}$ and satisfy the given work and span bounds.

We have provided a test for the correctness of these functions. Once you have written `map.mppcf` and `reduce.mppcf`, you can run `TestHarness.runmaptest ()` and `TestHarness.runreducetest ()` respectively to check the correctness of your functions. However, your function will be graded on correct usage of parallelism as well, while this only checks correctness.

A Testing your Implementation

KPCF$^+$ A REPL is available through `TopLevel.repl ()`, in which you can directly input **KPCF**$^+$ expressions and see the type and the value it evaluates to. Remember your input has to be a modal separated expression, otherwise the parser will reject your input outright. Here is an example interaction with the interpreter:

```
- TopLevel.repl();
->ret(z);
```

```
Statics: term has type Nat
Z
->ret(s(z));
Statics: term has type Nat
(Succ Z)
->ret(fn (x : C) ret(fn (y : D) ret(<x, y>)));
Statics: term has type (Arrow (C, (Arrow (D, (Prod (C, D))))))
(Lam ((x7, C) . (Ret (Lam ((y9, D) . (Ret (Prod (x7, y9))))))))
->(fn (x : nat) ret(s(x)))(s(z));
Statics: term has type Nat
(Succ (Succ Z))
```

A Testing harness can be accessed through `TestHarness.runalltests true`. It evaluates files listed in `tests/tests.sml`. These test files also serves as a syntax guide.

MPPCF There are many ways of testing your implementation. A REPL is available through `TopLevel.repl ()`, in which you may type **MPPCF** expressions and see the values that they evaluate to. Here is an example interaction with the interpreter:

```
- TopLevel.repl();
->ret(1)/ret(2);
0
->(fun f(x:nat):nat = ifz x {z=>ret(30) | s(x) => f(x)+ret(1)})(50);
80
->par y = {ret(1) & ret(2)} in split y as x1, x2 in ret(x1);
1
```

There is also a testing harness which you can access through `TestHarness.runalltests`. The test harness is located in `tests/tests.sml`.

Reference Implementation We have also included the solution to this assignment as a binary heap image. You can load it into SML/NJ by passing in the `@SMLload=refsol` flag. Your solutions should behave just like ours.

B Modal PCF with K Machines

Typing of Values and Expressions

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathtt{ret}(\, v \,) \div \tau}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \qquad \frac{\Gamma \vdash v : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(\, v \,) : \mathtt{nat}} \qquad \frac{\Gamma \vdash v : \mathtt{nat} \quad \Gamma \vdash e_1 \div \tau \quad \Gamma, x : \mathtt{nat} \vdash e_2 \div \tau}{\Gamma \vdash \mathtt{ifz}\{e_1; x \,.\, e_2\}(\, v \,) \div \tau}$$

$$\frac{\Gamma \vdash e \div \tau}{\Gamma \vdash \mathtt{comp}(\, e \,) : \tau \ \mathtt{comp}} \qquad \frac{\Gamma \vdash v : \tau_1 \ \mathtt{comp} \quad \Gamma, x : \tau_1 \vdash e \div \tau_2}{\Gamma \vdash \mathtt{bind}(\, v; x \,.\, e \,) \div \tau_2}$$

$$\frac{\Gamma, f : \tau_1 \rightharpoonup \tau_2, x : \tau_1 \vdash e \div \tau_2}{\Gamma \vdash \mathtt{fun}\{\tau_1; \tau_2\}(\, f \,.\, x \,.\, e \,) : \tau_1 \rightharpoonup \tau_2} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \rightharpoonup \tau_2 \quad \Gamma \vdash v_2 : \tau_1}{\Gamma \vdash \mathtt{ap}(\, v_1; v_2 \,) \div \tau_2}$$

Stacks and Safety

$$\frac{}{\epsilon \div \tau} \qquad \frac{x : \tau \vdash e \div \tau' \quad k \div \tau'}{k; x.e \div \tau}$$

Notice that typing for e does **not** carry a context Γ with it. This is not a shorthand or a mistake: since we evaluate only closed terms, x should be the only free variable in e. Be sure to implement this correctly.

$$\frac{k \div \tau \quad \cdot \vdash v : \tau}{k \triangleleft v \ \mathsf{ok}} \qquad \frac{k \div \tau \quad \cdot \vdash e \div \tau}{k \triangleright e \ \mathsf{ok}}$$

Dynamics

$$\frac{}{\epsilon \triangleleft v \ \mathsf{final}} \qquad \frac{}{k \triangleright \mathtt{ret}(\, v \,) \longmapsto k \triangleleft v} \qquad \frac{}{k; x.e_1 \triangleleft v \longmapsto k \triangleright [v/x]e_1}$$

$$\frac{}{k \triangleright \mathtt{bind}(\, \mathtt{comp}(\, e \,); x \,.\, e_1 \,) \longmapsto k; x.e_1 \triangleright e}$$

$$\frac{}{k \triangleright \mathtt{ifz}\{e_0; x \,.\, e_1\}(\, \mathtt{z} \,) \longmapsto k \triangleright e_0} \qquad \frac{}{k \triangleright \mathtt{ifz}\{e_0; x \,.\, e_1\}(\, \mathtt{s}(\, v \,) \,) \longmapsto k \triangleright [v/x]e_1}$$

$$\frac{}{k \triangleright \mathtt{ap}(\, \mathtt{fun}\{\tau_1; \tau_2\}(\, f \,.\, x \,.\, e \,); v_2 \,) \longmapsto k \triangleright [\mathtt{fun}\{\tau_1; \tau_2\}(\, f \,.\, x \,.\, e \,)e, v_2/f, x]e}$$

C The Language KPCF$^+$

Statics

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(\, v_1; v_2 \,) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash v : \tau_1 \times \tau_2 \quad \Gamma,\ x : \tau_1,\ y : \tau_2 \vdash e \mathrel{\dot{\sim}} \tau}{\Gamma \vdash \mathtt{split}[\tau](\, v; x, y \,.\, e \,) \mathrel{\dot{\sim}} \tau}$$

$$\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \mathtt{in}[\mathtt{l}]\{\tau_1; \tau_2\}(\, v \,) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash \mathtt{in}[\mathtt{r}]\{\tau_1; \tau_2\}(\, v \,) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash v : \tau_1 + \tau_2 \quad \Gamma,\ x : \tau_1 \vdash e_1 \mathrel{\dot{\sim}} \tau \quad \Gamma,\ y : \tau_2 \vdash e_2 \mathrel{\dot{\sim}} \tau}{\Gamma \vdash \mathtt{case}\{x \,.\, e_1; y \,.\, e_2\}(\, v \,) \mathrel{\dot{\sim}} \tau}$$

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}} \qquad \frac{\Gamma \vdash v : \mathtt{void}}{\Gamma \vdash \mathtt{abort}\{\tau\}(\, v \,) \mathrel{\dot{\sim}} \tau}$$

$$\frac{k \mathrel{\dot{\div}} \tau}{\Gamma \vdash \mathtt{cont}(\, k \,) : \mathtt{cont}(\, \tau \,)} \qquad \frac{\Gamma,\ x : \tau\, \mathtt{cont} \vdash e \mathrel{\dot{\sim}} \tau}{\Gamma \vdash \mathtt{letcc}\{\tau\}(\, x \,.\, e \,) \mathrel{\dot{\sim}} \tau} \qquad \frac{\Gamma \vdash v_1 : \tau' \quad \Gamma \vdash v_2 : \tau'\, \mathtt{cont}}{\Gamma \vdash \mathtt{throw}\{\tau\}(\, v_2; v_1 \,) \mathrel{\dot{\sim}} \tau}$$

Dynamics

$$\overline{k \triangleright \mathtt{split}[\tau](\, \langle v_1, v_2 \rangle; x, y \,.\, e \,) \longmapsto k \triangleright [v_1, v_2/x, y]e}$$

$$\overline{k \triangleright \mathtt{case}\{x \,.\, e_1; y \,.\, e_2\}(\, \mathtt{l} \cdot v \,) \longmapsto k \triangleright [v/x]e_1} \qquad \overline{k \triangleright \mathtt{case}\{x \,.\, e_1; y \,.\, e_2\}(\, \mathtt{r} \cdot v \,) \longmapsto k \triangleright [v/y]e_2}$$

$$\overline{k \triangleright \mathtt{letcc}\{\tau\}(\, x \,.\, e \,) \longmapsto k \triangleright [\mathtt{cont}(\, k \,)/x]e} \qquad \overline{k \triangleright \mathtt{throw}\{\tau\}(\, \mathtt{cont}(\, k' \,); v \,) \longmapsto k' \triangleleft v}$$

There is no rule for the elimination form of `void`. Notice how simple defining dynamics are: you only need to take care for the elimination forms. Introduction forms are naturally taken care of through modal separation.