FUN — Untyped — Environment

Grigore Roşu and Traian Florin Şerbănuţă ({grosu,tserban2}@illinois.edu)
University of Illinois at Urbana-Champaign

MODULE MY-PATTERN-MATCHING

SYNTAX *KItem* ::= `#myMap` (*MyMap*)

SYNTAX *Bool* ::= `#myIsMapInK` (*K*) [function]

SYNTAX *MyMap* ::= `#myGetMapFromK` (*K*) [function]

RULE `#myIsMapInK ( #myMap (—))` ⇒ `true`

RULE `#myIsMapInK (—)` ⇒ `false` [owise]

RULE `#myGetMapFromK ( #myMap (M))` ⇒ M

SYNTAX *Bool* ::= `myIsMatching` (*KList*, *KList*) [function]

RULE `myIsMatching (Ps, Ks)` ⇒ `#myIsMapInK ( #myPatternMatch (Ps, Ks))` $=_K$ `true`

SYNTAX *MyMap* ::= `myGetMatching` (*KList*, *KList*) [function]

SYNTAX *KItem* ::= `#myPatternMatch` (*KList*, *KList*) [function]

RULE `myGetMatching (Ps, Ks)` ⇒ `#myGetMapFromK ( #myPatternMatch (Ps, Ks))`

SYNTAX *KList* ::= `myDecomposeMatching` (*KList*, *KList*) [function]

SYNTAX *KItem* ::= `myDoneDecomposeMatching` (*Variable*, *K*)

SYNTAX *KItem* ::= `#unmatchableError` (*KList*, *KList*)

RULE `myDecomposeMatching (L(Ps), L(Ks))` ⇒ `myDecomposeMatching (Ps, Ks)` requires $isVariable(L(Ps)) \neq_K$ `true`

RULE `myDecomposeMatching (P, K)` ⇒ `myDoneDecomposeMatching (P, K)`

RULE `myDecomposeMatching (P, Ps, K, Ks)` ⇒ `myDecomposeMatching (P, K), myDecomposeMatching (Ps, Ks)` requires $\neg_{Bool}$ `emptyKList (Ps)`

RULE `myDecomposeMatching (Ps, Ks)` ⇒ `#if Ps ==KList Ks #then #else #unmatchableError (Ps, Ks) #fi` [owise]

SYNTAX *Bool* ::= `emptyKList` (*KList*) [function]

RULE `emptyKList ()` ⇒ `true`

RULE `emptyKList (K, Ks)` ⇒ `false`

SYNTAX *Variable* ::= *Id*

SYNTAX *KItem* ::= `#myPatternMatch` (*KList*, *KList*) [function]

RULE `#myPatternMatch (Ps, Ks)` ⇒ `#myPatternMatch1 ( myDecomposeMatching (Ps, Ks), .MyMap)`

SYNTAX *KItem* ::= `#myPatternMatch1` (*KList*, *MyMap*) [function]

SYNTAX *KItem* ::= `#nonlinearPatternError`

RULE `#myPatternMatch1 ( myDoneDecomposeMatching (X, K), Ps, M)` ⇒ `#myPatternMatch1 (Ps, (M, X |-> K))` requires $\neg_{Bool}(X$ `in keys` $(M))$

RULE `#myPatternMatch1 ( #unmatchableError (P, K), —, —)` ⇒ `#unmatchableError (P, K)`

RULE `#myPatternMatch1 ( myDoneDecomposeMatching (X, K), Ps, M)` ⇒ `#nonlinearPatternError` requires X `in keys` (M)

RULE `#myPatternMatch1 (, M)` ⇒ `#myMap (M)`

END MODULE

Abstract

This is the 𝕂 semantic definition of the untyped FUN language. FUN is a pedagogical and research language that captures the essence of the functional programming paradigm, extended with several features often encountered in functional programming languages. Like many functional languages, FUN is an expression language, that is, everything, including the main program, is an expression. Functions can be declared anywhere and are first class values in the language. FUN is call-by-value here, but it has been extended (as student homework assignments) with other parameter-passing styles. To make it more interesting and to highlight some of 𝕂's strengths, FUN includes the following features:

- The basic builtin data-types of integers, booleans and strings.
- Builtin lists, which can hold any elements, including other lists. Lists are enclosed in square brackets and their elements are comma-separated; e.g., `[1,2,3]`.
- User-defined data-types, by means of constructor terms. Constructor names start with a capital letter (while any other identifier in the language starts with a lowercase letter), and they can be followed by an arbitrary number of comma-separated arguments enclosed in parentheses; parentheses are not needed when the constructor takes no arguments. For example, `Pair(5,7)` is a constructor term holding two numbers, `Cons(1,Cons(2,Cons(3,Nil)))` is a list-like constructor term holding 3 elements, and `Tree(Tree(Leaf(1), Leaf(2)), Leaf(3))` is a tree-like constructor term holding 3 elements. In the untyped version of the FUN language, no type checking or inference is performed to ensure that the data constructors are used correctly. The execution will simply get stuck when they are misused. Moreover, since no type checking is performed, the data-types are not even declared in the untyped version of FUN.
- Functions and `let`/`letrec` binders can take multiple space-separated arguments, but these are desugared to ones that only take one argument, by currying. For example, the expressions

  ```
  fun x y -> x y
  let x y = y in x
  ```

  are desugared, respectively, into the following expressions:

  ```
  fun x -> fun y -> x y
  let x = fun y -> y in x
  ```

- Functions can be defined using pattern matching over the available data-types. For example, the program

  ```
  letrec max = fun [h] -> h
               |   [h|t] -> let x = max t
                            in  if h > x then h else x
  in max [1, 3, 5, 2, 4, 0, -1, -5]
  ```

  defines a function `max` that calculates the maximum element of a non-empty list, and the function

  ```
  letrec ack = fun Pair(0,n) -> n + 1
               |   Pair(m,0) -> ack Pair(m - 1, 1)
               |   Pair(m,n) -> ack Pair(m - 1, ack Pair(m, n - 1))
  in ack Pair(2,3)
  ```

  calculates the Ackermann function applied to a particular pair of numbers. Patterns can be nested. Patterns can currently only be used in function definitions, and not directly in `let`/`letrec` binders. For example, this is not allowed:

  ```
  letrec Pai(x,y) = Pair(1,2) in x+y
  ```

  But this is allowed:

  ```
  let f Pair(x,y) = x+y in f Pair(1,2)
  ```

  because it is first reduced to

  ```
  let f = fun Pair(x,y) -> x+y in f Pair(1,2)
  ```

  by uncurrying of the `let` binder, and pattern matching is allowed in function arguments.
- We include a `callcc` construct, for two reasons: first, several functional languages support this construct; second, some semantic frameworks have difficulties defining it. Not 𝕂.
- Finally, we include mutables by means of referencing an expression, getting the reference of a variable, dereferencing and assignment. We include these for the same reasons as above: there are languages which have them, and they are not easy to define in some semantic frameworks.

Like in many other languages, some of FUN's constructs can be desugared into a smaller set of basic constructs. We do that as usual, using macros, and then we only give semantics to the core constructs.

Note: We recommend the reader to first consult the dynamic semantics of the LAMBDA++ language in the first part of the K Tutorial. To keep the comments below small and focused, we will not re-explain functional or 𝕂 features that have already been explained in there.

Syntax

MODULE FUN-UNTYPED-SYNTAX

FUN is an expression language. The constructs below fall into several categories: names, arithmetic constructs, conventional functional constructs, patterns and pattern matching, data constructs, lists, references, and call-with-current-continuation (callcc). The arithmetic constructs are standard; they are present in almost all our 𝕂 language definitions. The meaning of FUN's constructs are discussed in more depth when we define their semantics in the next module.

The Syntactic Constructs

We start with the syntactic definition of FUN names. We have several categories of names: ones to be used for functions and variables, others to be used for data constructors, others for types and others for type variables. We will introduce them as needed, starting with the former category. We prefer the names of variables and functions to start with lower case letters. We take the freedom to tacitly introduce syntactic lists/sequences for each nonterminal for which we need them:

SYNTAX *Name* ::= $Token\{[a-z][_a-zA-Z0-9]*\}$ [notInRules]

SYNTAX *Names* ::= $List\{Name, “,”\}$

Expression constructs will be defined throughtout the syntax module. Below are the very basic ones, namely the builtins, the names, and the parentheses used as brackets for grouping. Lists of expressions are declared strict, so all expressions in the list get evaluated whenever the list is on a position which can be evaluated:

SYNTAX *Exp* ::= *Int*
  | *Bool*
  | *String*
  | *Name*
  | (*Exp*) [bracket]

SYNTAX *Exps* ::= $List\{Exp, “,”\}$ [strict]

We next define the syntax of arithmetic constructs, together with their relative priorities and left-/non-associativities. We also tag all these rules with a new tag, “arith”, so we can more easily define global syntax priirities later (at the end of the syntax module).

SYNTAX *Exp* ::= *Exp* `*` *Exp* [strict, arith]
  | *Exp* `/` *Exp* [strict, arith]
  | *Exp* `%` *Exp* [strict, arith]
  | *Exp* `+` *Exp* [strict, arith]
  | *Exp* `^` *Exp* [strict, arith]
  | *Exp* `-` *Exp* [strict, prefer, arith]
  | *Exp* `<` *Exp* [strict, arith]
  | *Exp* `<=` *Exp* [strict, arith]
  | *Exp* `>` *Exp* [strict, arith]
  | *Exp* `>=` *Exp* [strict, arith]
  | *Exp* `==` *Exp* [strict, arith]
  | *Exp* `!=` *Exp* [strict, arith]
  | `!` *Exp* [strict, arith]
  | *Exp* `&&` *Exp* [strict(1), arith]
  | *Exp* `||` *Exp* [strict(1), arith]

The conditional construct has the expected evaluation strategy, stating that only the first argument is evaluate:

SYNTAX *Exp* ::= `if` *Exp* `then` *Exp* `else` *Exp* [strict(1)]

FUN's builtin lists are formed by enclosing comma-separated sequences of expressions (i.e., terms of sort *Exps*) in square brackets. The list constructor `cons` adds a new element to the top of the list, `head` and `tail` get the first element and the tail sublist of a list if they exist, respectively, and get stuck otherwise, and `null??` tests whether a list is empty or not; syntactically, these are just expression constants. In function patterns, we are also going to allow patterns following the usual head/tail notation; for example, the pattern $[x_1, ..., x_n|t]$ binds x_1, ..., x_n to the first elements of the matched list, and `t` to the list formed with the remaining elements. We define list patterns as ordinary expression constructs, although we will make sure that we do not give them semantics if they appear in any other place then in a function case pattern.

SYNTAX *Exp* ::= [*Exps*] [strict]
  | `cons`
  | `head`
  | `tail`
  | `null?`
  | [*Exps* `|` *Exp*]

Data constructors start with capital letters and they may or may not have arguments. We need to use the attribute “prefer” to make sure that, e.g., `Cons(a)` parses as constructor `Cons` with argument `a`, and not as the expression `Cons` (because constructor names are also expressions) regarded as a function applied to the expression `a`. Also, note that the constructor is strict in its second argument, because we want to evaluate its arguments but not the constuctor name itself.

SYNTAX *ConstructorName* ::= $Token\{[A-Z][a-zA-Z0-9]*\}$ [notInRules]

SYNTAX *Exp* ::= *ConstructorName*
  | *ConstructorName*(*Exps*) [prefer, strict(2)]

A function is essentially a “|”-separated ordered sequence of cases, each case of the form “`pattern -> expression`”, preceded by the language construct `fun`. Patterns will be defined shortly, both for the builtin lists and for user-defined constructors. Recall that the syntax we define in 𝕂 is not meant to serve as a ultimate parser for the defined language, but rather as a convenient notation for 𝕂 abstract syntax trees, which we prefer when we write the semantic rules. It is therefore often the case that we define a more “generous” syntax than we want to allow programs to use. We do it here, too. Specifically, the syntax of *Cases* below allows any expressions to appear as pattern. This syntactic relaxation permits many wrong programs to be parsed, but that is not a problem because we are not going to give semantics to wrong combinations, so those programs will get stuck; moreover, our type inferencer will reject those programs anyway. Function application is just concatenation of expressions, without worrying about type correctness. Again, the type system will reject type-incorrect programs.

SYNTAX *Exp* ::= `fun` *Cases*
  | *Exp* *Exp* [strict]

SYNTAX *Case* ::= *Exp* `->` *Exp*

SYNTAX *Cases* ::= $List\{Case, “|”\}$

The `let` and `letrec` binders have the usual syntax and functional meaning. We allow multiple `and`-separated bindings. Like for the function cases above, we allow a more generous syntax for the left-hand sides of bindings, noting that the semantics will get stuck on incorrect bindings and that the type system will reject those programs.

SYNTAX *Exp* ::= `let` *Bindings* `in` *Exp*
  | `letrec` *Bindings* `in` *Exp* [prefer]

SYNTAX *Binding* ::= *Exp* `=` *Exp*

SYNTAX *Bindings* ::= $List\{Binding, “and”\}$

References are first class values in FUN. The construct `ref` takes an expression, evaluates it, and then it stores the resulting value at a fresh location in the store and returns that reference. Syntactically, `ref` is just an expression constant. The construct `&` takes a name as argument and evaluates to a reference, namely the store reference where the variable passed as argument stores its value; this construct is a bit controversial and is further discussed in the environment-based semantics of the FUN language, where we desugar `ref` to it. The construct `@` takes a reference and evaluates to the value stored there. The construct `:=` takes two expressions, the first expected to evaluate to a reference; the value of its second argument will be stored at the location to which the first points (the old value is thus lost). Finally, since expression evaluation now has side effects, it makes sense to also add a sequential composition construct, which is sequentially strict. This evaluates to the value of its second argument; the value of the first argument is lost (which has therefore been evaluated only for its side effects.

SYNTAX *Exp* ::= `ref`
  | `&` *Name*
  | `@` *Exp* [strict]
  | *Exp* `:=` *Exp* [strict]
  | *Exp* `;` *Exp* [strict(1)]

Call-with-current-continuation, named `callcc` in FUN, is a powerful control operator that originated in the Scheme programming language, but it now exists in many other functional languages. It works by evaluating its argument, expected to evaluate to a function, and by passing the current continuation, or evaluation context (or computation, in 𝕂 terminology), as a special value to it. When/If this special value is invoked, the current context is discarded and replaced with the one held by the special value and the computation continues from there. It is like taking a snapshot of the execution context at some moment in time and then, when desired, being able to get back in time to that point. If you like games, it is like saving the game now (so you can work on your homework!) and then continuing the game tomorrow or whenever you wish. To issustrate the strength of `callcc`, we also allow exceptions in FUN by means of a conventional `try-catch` construct, which will desugar to `callcc`. We also need to introduce the special expression contant `throw`, but we need to use it as a function argument name in the desugaring macro, so we define it as a name instead of as an expression constant:

SYNTAX *Exp* ::= `callcc`
  | `try` *Exp* `catch` (*Name*)*Exp*

SYNTAX *Name* ::= $Token\{”throw”\}$

Finally, FUN also allows polymorphic datatype declarations. These will be useful when we define the type system later on.

SYNTAX *Exp* ::= `datatype` *Type* `=` *TypeCases* *Exp*

We next need to define the syntax of types and type cases that appear in datatype declarations.

Like in many functional languages, type parameters/variables in user-defined types are quoted identifiers.

SYNTAX *TypeVar* ::= $Token\{[\backslash'][a-z][_a-zA-Z0-9]*\}$ [notInRules]

SYNTAX *TypeVars* ::= $List\{TypeVar, “,”\}$

Types can be basic types, function types, or user-defined parametric types. In the dynamic semantics we are going to simply ignore all the type declations, so here the syntax of types below is only useful for generating the desired parser. To avoid syntactic ambiguities with the arrow construct for function cases, we use the symbol `-->` as a constructor for function types:

SYNTAX *TypeName* ::= $Token\{[a-z][_a-zA-Z0-9]*\}$ [notInRules]

SYNTAX *Type* ::= `int`
  | `bool`
  | `string`
  | *Type* `->` *Type*
  | (*Type*) [bracket]
  | *TypeVar*
  | *TypeName* [klabel('TypeName), onlyLabel]
  | *Type* *TypeName* [klabel('Type-TypeName), onlyLabel]
  | (*Types*)*TypeName* [prefer]

SYNTAX *Types* ::= $List\{Type, “,”\}$

SYNTAX *TypeCase* ::= *ConstructorName*
  | *ConstructorName*(*Types*)

SYNTAX *TypeCases* ::= $List\{TypeCase, “|”\}$

Additional Priorities

Desugaring macros

We desugar the list non-constructor operations to functions matching over list patterns. In order to do that we need some new variables; for those, we follow the same convention like in the 𝕂 tutorial, where we added them as new identifier constructs starting with the character $, so we can easily recognize them when we debug or trace the semantics.

SYNTAX *Name* ::= $Token\{”\$h”\}$
  | $Token\{”\$t”\}$

RULE `head` ⇒ `fun [$h | $t] -> $h` [macro]

RULE `tail` ⇒ `fun [$h | $t] -> $t` [macro]

RULE `null?` ⇒ `fun [`$\bullet_{Exps}$`] -> true| [$h | $t] -> false` [macro]

Multiple-head list patterns desugar into successive one-head patterns:

RULE `[E, Es | T]` ⇒ `[E | [Es | T]]` requires $Es \neq_K \bullet_{Exps}$ [macro]

Uncurrying of multiple arguments in functions and binders:

RULE `P1 P2 -> E` ⇒ `P1 -> fun P2 -> E` [macro]

RULE `F P = E` ⇒ `F = fun P -> E` [macro]

We desugar the `try-catch` construct into callcc:

SYNTAX *Name* ::= $Token\{”\$k”\}$
  | $Token\{”\$v”\}$

RULE `try E catch (X)E'` ⇒ `callcc ( fun $k -> ( fun throw -> E) ( fun X -> $k E'))` [macro]

For uniformity, we reduce all types to their general form:

RULE $'TypeName(Tn)$ ⇒ $(\bullet_{TypeVars})Tn$ [macro]

RULE $'Type-TypeName(T, Tn)$ ⇒ $(T)Tn$ [macro]

The dynamic semantics ignores all the type declarations:

RULE `datatype T = TCs E` ⇒ E [macro]

END MODULE

Semantics

The semantics below is environment-based. A substitution-based definition of FUN is also available, but that drops the & construct as explained above.

MODULE FUN-UNTYPED

Configuration

The `k`, `env`, and `store` cells are standard (see, for example, the definition of LAMBDA++ or IMP++ in the first part of the 𝕂 tutorial).

CONFIGURATION:

⟨ ⟨ $PGM ⟩k ⟨ .MyMap ⟩env ⟨ .MyMap ⟩store ⟩T

Values and results

We only define integers, Booleans and strings as values here, but will add more values later.

SYNTAX *Val* ::= *Int*
  | *Bool*
  | *String*

SYNTAX *Vals* ::= $List\{Val, “,”\}$

SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

Lookup

RULE ⟨ X ⇒ V …⟩k ⟨… X `|->` L …⟩env ⟨… L `|->` V …⟩store

Arithmetic expressions

RULE $I1$ `*` $I2$ ⇒ $I1 *_{Int} I2$

RULE $I1$ `/` $I2$ ⇒ $I1 \div_{Int} I2$ requires $I2 \neq_K 0$

RULE $I1$ `%` $I2$ ⇒ $I1 \%_{Int} I2$ requires $I2 \neq_K 0$

RULE $I1$ `+` $I2$ ⇒ $I1 +_{Int} I2$

RULE $S1$ `^` $S2$ ⇒ $S1 +_{String} S2$

RULE $I1$ `-` $I2$ ⇒ $I1 -_{Int} I2$

RULE $I1$ `<` $I2$ ⇒ $I1 <_{Int} I2$

RULE $I1$ `<=` $I2$ ⇒ $I1 \leq_{Int} I2$

RULE $I1$ `>` $I2$ ⇒ $I1 >_{Int} I2$

RULE $I1$ `>=` $I2$ ⇒ $I1 \geq_{Int} I2$

RULE $V1$ `==` $V2$ ⇒ $V1 =_K V2$

RULE $V1$ `!=` $V2$ ⇒ $V1 \neq_K V2$

RULE `!` T ⇒ $\neg_{Bool}(T)$

RULE `true &&` E ⇒ E

RULE `false && —` ⇒ `false`

RULE `true || —` ⇒ `true`

RULE `false ||` E ⇒ E

Conditional

RULE `if true then E else —` ⇒ E

RULE `if false then — else E` ⇒ E

Lists

We have already declared the syntactic list of expressions strict, so we can assume that all the elements that appear in a FUN list are evaluated. The only thing left to do is to state that a list of values is a value itself, that is, that the list square-bracket construct is indeed a constructor, and to give the semantics of `cons`. Since `cons` is a builtin function and is expected to take two arguments, we have to also state that `cons` itself is a value (specifically, a function/closure value, but we do not need that level of detail here), and also that `cons` applied to a value is a value (specifically, it would be a function/closure value that expects the second, list argument):

SYNTAX *Val* ::= `cons`
  | [*Vals*]

RULE $isVal($ `cons` $V)$ ⇒ `true`

RULE `cons V [Vs]` ⇒ `[V, Vs]`

Data Constructors

Constructors take values as arguments and produce other values:

SYNTAX *Val* ::= *ConstructorName*
  | *ConstructorName*(*Vals*)

Functions and Closures

Like in the environment-based semantics of LAMBDA++ in the first part of the 𝕂 tutorial, functions evaluate to closures. A closure includes the current environment besides the function contents; the environment will be used at execution time to lookup all the variables that appear free in the function body (we want static scoping in FUN).

SYNTAX *Val* ::= `closure` (*MyMap*, *Cases*)

RULE ⟨ `fun Cases` ⇒ `closure (ρ, Cases)` …⟩k ⟨ ρ ⟩env

Note: The reader may want to get familiar with how the pre-defined pattern matching works before proceeding. The best way to do that is to consult `k/include/modules/pattern-matching.k`.

To set up the pattern matching mechanism we need to specify what K terms act as variables (for pattern matching, substitution, etc.). This is currently done my subsorting those terms to the builtin `Variable` sort. In our case, we only want to allow the `Name` identifiers to act as variables for pattern matching; note that the `ConstructorName` identifiers are *not* variables (they construct data values):

SYNTAX *Variable* ::= *Name*

We distinguish two cases when the closure is applied. If the first pattern matches, then we pick the first case: switch to the closed environment, get the matching map and bind all its variables, and finally evaluate the function body of the first case, making sure that the environment is properly recovered afterwards. If the first pattern does not match, then we drop it and thus move on to the next one.

RULE ⟨ `closure (ρ, P -> E | —) V` ⇒ `bindTo ( myGetMatching (P, V))` ↷ E ↷ `env (ρ')` …⟩k ⟨ ρ' ⇒ ρ ⟩env requires `myIsMatching (P, V)`

RULE `closure (—, P -> — | Cs) V` ⇒ `Cs` requires $\neg_{Bool}$ `myIsMatching (P, V)`

Besides the generic decomposition rules for patterns and values, we also want to allow `[head|tail]` matching for lists, so we add the following custom pattern decomposition rule:

RULE `myDecomposeMatching ([H | T], [V, Vs])` ⇒ `myDecomposeMatching ((H, T), (V, [Vs]))`

Let and Letrec

To highlight the similarities and differences between `let` and `letrec`, we prefer to give them direct semantics instead of to desugar them like in LAMBDA. See the formal definitions of `bindTo`, `bind`, and `assignTo` at the end of this module. Informally, `bindTo(`Xs`,`Es`)` first evaluates the expressions $Es \in Exps$ in the current environment (i.e., it is strict in its second argument), then it binds the variables in $Xs \in Names$ to new locations and adds those bindings to the environment, and finally writes the values previously obtained after evaluating the expressions Es to those new locations; `bind(`Xs`)` does only the bindings of Xs to new locations and adds those bindings to the environment; and `assignTo(`Xs`,`Es`)` evaluates the expressions Es in the current environment and then it writes the resulting values to the locations to which the variables Xs are already bound to in the environment.

Therefore, “`let` Xs`=`Es `in` E” first evaluates Es in the current environment, then adds new bindings for Xs to fresh locations in the environment, then writes the values of Es to those locations, and finally evaluates E in the new environment, making sure that the environment is properly recovered after the evaluation of E. On the other hand, `letrec` does the same things but in a different order: it first adds new bindings for Xs to fresh locations in the environment, then it evaluates Es in the new environment, then it writes the resulting values to their corresponding locations, and finally it evaluates E and recovers the environment. The crucial difference is that the expressions Es now see the locations of the variables Xs in the environment, so if they are functions, which is typically the case with `letrec`, their closures will encapsulate in their environments the bindings of all the bound variables, including themselves (thus, we may have a closure value stored at location L, whose environment contains a binding of the form $F \mapsto L$; this way, the closure can invoke itself).

RULE ⟨ `let Bs in E` ⇒ `bindTo ( names (Bs), exps (Bs))` ↷ E ↷ `env (ρ)` …⟩k ⟨ ρ ⟩env

RULE ⟨ `letrec Bs in E` ⇒ `bind ( names (Bs))` ↷ `assignTo ( names (Bs), exps (Bs))` ↷ E ↷ `env (ρ)` …⟩k ⟨ ρ ⟩env

Recall that our syntax allows `let` and `letrec` to take any expression in place of its binding. This allows us to use the already existing function application construct to bind names to functions, such as, e.g., “`let x y = y in ...`”. The desugaring macro in the syntax module uncurries such declarations, and then the semantic rules above only work when the remaining bindings are identifiers, so the semantics will get stuck on programs that misuse the `let` and `letrec` binders.

References

The semantics of references is self-explanatory, except maybe for the desugaring rule of `ref`, which is further discussed. Note that &X grabs the location of X from the environment. Sequential composition, which is needed only to accumulate the side effects due to assignments, was strict in the first argument. Once evaluated, its first argument is simply discarded:

SYNTAX *Name* ::= $Token\{”\$x”\}$

RULE `ref` ⇒ `fun $x -> & $x` [macro]

RULE ⟨ `& X` ⇒ L …⟩k ⟨… X `|->` L …⟩env

RULE ⟨ `@ L` ⇒ V …⟩k ⟨… L `|->` V …⟩store

RULE ⟨ `L := V` ⇒ V …⟩k ⟨… L `|->` (— ⇒ V) …⟩store

RULE `V ; E` ⇒ E

The desugaring rule of `ref` (first rule above) works because `&` takes a variable and returns its location (like in C). Note that some “pure” functional programming researchers strongly dislike the `&` construct, but favor `ref`. We refrain from having a personal opinion on this issue here, but support `&` in the environment-based definition of FUN because it is, technically speaking, more powerful than `ref`. From a language design perspective, it would be equally easy to drop `&` and instead give a direct semantics to `ref`. In fact, this is precisely what we do in the substitution-based definition of FUN, because there appears to be no way to give a substitution-based definition to the `&` construct.

Callcc

As we know it from the LAMBDA++ tutorial, call-with-current-continuation is quite easy to define in 𝕂. We first need to define a special value wrapping an execution context, that is, an environment saying where the variables should be looked up, and a computation structure saying what is left to execute (in a substitution-based definition, this special value would be even simpler, as it would only need to wrap the computation structure—see, for example, the substitution-based semantics of LAMBDA++ in the the first part of the 𝕂 tutorial, or the substitution-based definition of FUN). Then `callcc` creates such a value containing the current environment and the current remaining computation, and passes it to its argument function. When/If invoked, the special value replaces the current execution context with its own and continues the execution normally.

SYNTAX *Val* ::= `callcc`
  | `cc` (*MyMap*, *K*)

RULE ⟨ `callcc V` ↷ K ⇒ `V cc (ρ, K)` ⟩k ⟨ ρ ⟩env

RULE ⟨ `cc (ρ, K) V` ↷ — ⇒ V ↷ K ⟩k ⟨ — ⇒ ρ ⟩env

Auxiliary operations

Environment recovery. The environment recovery operation is the same as for the LAMBDA++ language in the 𝕂 tutorial and many other languages provided with the 𝕂 distribution. The first “anywhere” rule below shows an elegant way to achieve the benefits of tail recursion in 𝕂.

SYNTAX *KItem* ::= `env` (*MyMap*)

RULE `env (—)` ⇒ $\bullet_K$ ↷ `env (—)` [anywhere]

RULE ⟨ — ↷ `env (ρ)` ⇒ $\bullet_K$ …⟩k ⟨ — ⇒ ρ ⟩env [structural]

`bindTo`, `bind` and `assignTo`. The meaning of these operations has already been explained when we discussed the `let` and `letrec` language constructs above.

SYNTAX *KItem* ::= `bindTo` (*Names*, *Exps*) [strict(2)]
  | `bindTo` (*MyMap*)
  | `bind` (*Names*)

RULE `bindTo (Xs, Vs)` ⇒ `bindTo ( myGetMatching (Xs, Vs))`

RULE `bindTo ( .MyMap)` ⇒ $\bullet_K$ [structural]

RULE ⟨ `bindTo (X |-> V, —` ⇒ `.MyMap)` …⟩k ⟨ ρ ⇒ $\rho[X$ `<-` $L]$ ⟩env ⟨… `.MyMap` ⇒ L `|->` V …⟩store requires `fresh (L)` [structural]

RULE `bind (`$\bullet_{Names}$`)` ⇒ $\bullet_K$ [structural]

RULE ⟨ `bind (X, Xs` ⇒ `Xs)` …⟩k ⟨ ρ ⇒ $\rho[X$ `<-` $L]$ ⟩env requires `fresh (L)` [structural]

SYNTAX *KItem* ::= `assignTo` (*Names*, *Exps*) [strict(2)]

RULE ⟨ `assignTo (`$\bullet_{Names}, \bullet_{Vals}$`)` ⇒ $\bullet_K$ …⟩k [structural]

RULE ⟨ `assignTo (X, Xs` ⇒ `Xs, V, Vs` ⇒ `Vs)` …⟩k ⟨… X `|->` L …⟩env ⟨… `.MyMap` ⇒ L `|->` V …⟩store [structural]

Getters. The following auxiliary operations extract the list of identifiers and of expressions in a binding, respectively.

SYNTAX *Names* ::= `names` (*Bindings*) [function]

RULE `names (`$\bullet_{Bindings}$`)` ⇒ $\bullet_{Names}$

RULE `names (X = —and Bs)` ⇒ `(X, names (Bs))`

SYNTAX *Exps* ::= `exps` (*Bindings*) [function]

RULE `exps (`$\bullet_{Bindings}$`)` ⇒ $\bullet_{Exps}$

RULE `exps (— = Eand Bs)` ⇒ `E, exps (Bs)`

END MODULE