FUN — Untyped — Environment

Grigore Roșu and Traian Florin Șerbănuță {grosu,tserban2}@illinois.edu
University of Illinois at Urbana-Champaign

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. FUN is dynamically typed, has first-class functions, lexical scoping, recursion, and a few basic types, but no type or scope declarations. Functions can be declared anywhere and can have either names or be anonymous. FUN is call-by-value but it also supports call-by-name and call-by-need parameter passing. It has a few basic types, such as booleans and integers, and it has a few built-in operations, but no user-defined ones. FUN includes the following features:

- Arithmetic, boolean, and relational expressions, including short-circuit, but not enclosed in square brackets and their elements are comma-separated, as in `[1,2,3]`.
- Conditionals, sequential composition, and conjunctive cases. Conjunctive cases 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. Conjunctive cases are matched against expressions in `case` statements, as in `case Cons(1,Cons(2,Nil)) of Cons(X,Cons(Y,Nil)) => X+Y`. The constructor `Nil` is used like a constant, and `Cons(1,Cons(2,Nil))` is used like a function call. The `case` statement is also used to match against the arguments of a function call, to ensure that the data constructors are used correctly. The constructor will simply get stuck when they are misused (for example, `Cons(1,2,3)` will get stuck), and the `case` statement will get stuck when no pattern matches (for example, `case 1 of Nil => 0`).
- Functions and `letrec`s (which are functions in which multiple mutually recursive functions are designated to start the ball) take one argument, by currying. For example, the expression

```
letrec f(x) = x + 1
   and g(x,y) = x * y
in f(g(2,3))
```

  are desugared, respectively, into the following expressions:

```
fun x -> fun y -> x + 1
let x = fun y -> x * y in
```

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

```
letrec map = fun [h|t] -> h :: map(t)
         | [] -> [];
in map(fun(x) -> x + 1, [1,2,3])
```

  defines a function that takes a function and a list and applies the function to each element of the list.

```
letrec add = fun Pair(0,Y) -> Y
         | Pair(S(X),Y) -> S(add(Pair(X,Y)));
in add(Pair(S(S(0)),S(S(S(0)))))
```

  calculates the 5-elements function applied to a particular pair of numbers. Patterns can be nested. Patterns can currently only be variables, constructors, or lists (which are desugared into `Cons` and `Nil` for the purpose). For example, this is also allowed:

```
letrec f(a,a) = a; f(a,b) = 0;
in f(1,1)
```

  But this is allowed:

```
let f = fun(a,a) -> a; a; f(a,b) = 0;
in f(1,1)
```

  because in first instead of

```
let f = fun Pair(a,a) -> a; a; f(a,b) = 0;
```

  by associating all the `let` binder, and pattern matching is allowed in function arguments.
- Records, a call/cc construct, but no exceptions. Key several functional languages support the construct, record, exist records, functions have difficulties defining it. Not S.
- Imperative features, including assignments and sequences, getting the references of a variable, dereferencing and assignment. We include them for the sake of completeness, but we do not recommend their use. Instead, we advise the user to strive to define as many semantic features as possible.

Note. We recommend the reader to first consult the dynamic semantics of the CLAIMED++ language in the first part of the K tutorial. Knowing the semantic basics and each feature, we will not re-explain functionality if it features that have already been explained before.

Syntax

MODULE FUN-UNTYPED-SYNTAX

FUN is an expression language. The constructs below fall into several categories: names, arithmetic constructs, conventional boolean constructs, optional and pattern matching, data constructs, lists, conventional control constructs, and miscellaneous constructs. The arithmetic constructs are standard, they are present in almost all (or all) language definitions. The remaining constructs are discussed in more detail in the sections where they are defined.

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 we need them. Names are defined as identifiers, which are sequences of letters, digits, and underscores, starting with a letter. We take the freedom to tacitly introduce syntactic list-manipulation for each enumerated list which we need them.

```
syntax Name ::= r"[a-zA-Z][a-zA-Z0-9_]*" [notInRules]

syntax Names ::= List{Name,","}
```

Expression constructs will be defined throughout the syntax module. Below are the very basic ones, namely the boolean, the integer, and the identifier expressions. Identifiers are also known as variables. Their semantics is discussed later. Note that the identifier declared in the list is a variation which has the syntax:

```
syntax Exp ::= Int
             | Bool
             | Id
             | "(" Exp ")" [bracket]

syntax Exps ::= List{Exp,","} [klabel]
```

We next define the syntax of conditionals and sequences. Conditionals are the usual if-then-else, but we also allow the else branch to be missing, in which case it is assumed to be the unit value, `()`. Sequences are also the usual ones, but we allow the last expression to be followed by a semicolon, which is also assumed to be the unit value, `()`. Sequences are right-associative.

```
syntax Exp ::= "if" Exp "then" Exp "else" Exp [strict(1)]
```

Unlike in most languages, we do not require the else branch to be present. The reason is that we want to be able to write conjunctive cases, which we discuss later, without having to write the else branch. Not requiring the else branch will not cause any ambiguities, because we will require the then branch to be enclosed in curly braces, which will also be required for sequences. Thus, we can write `if 1 then { 2; 3 }` and `if 1 then { 2; 3 } else 4`, but not `if 1 then 2; 3 else 4`.

```
syntax Exp ::= "if" Exp "then" Exp
             | "if" Exp "then" Exp "else" Exp [strict(1)]

syntax Exp ::= "{" Exp "}"
             | "{" Exp ";" "}"
             | "{" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" "}"
             > right: "{" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" Exp ";" "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" "}"
             | "{" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp ";" Exp "}"
```

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

```
syntax Exp ::= Exp "if Exp Then Exp Else" Exp [strict(2)]
```

Unlike most languages, we allow a conjunctive sequence of statements (i.e., series of case keys) to begin with a boolean. The first statement (case) adds a new element to the top of the list, Seed: each (sic!) per the first element and the rest of the list. Each case is then applied to the Seed and the rest of the list. Conjunctive cases are matched in order. Whenever a case does not match, the next conjunctive case is tried. Conjunctive cases are also grouping in other places following the same bracketed semicolon. For example, the pattern `{1,2,3,4,5}` binds `1` to `X`, to the first element of the matched list, and to the rest of the list, `{2,3,4,5}`, to `Y`. Conjunctive cases are also used in `case` statements, as in `case 1 of 1 => 2 | 3 => 4`. Note that in this case there is no need to have curly braces to group the cases, as there is no ambiguity.

```
syntax Exp ::= Cases
syntax Cases ::= Case
               | Case Cases
syntax Case ::= "|" Exp "=>" Exp
```

The `Case` and `Cases` nonterminals are used for the syntax of conjunctive cases. Note that we do not need to write explicitly the semicolon between the boolean and the rest of the case, because it is already part of the syntax of `Exp`.

The `Case` and `Cases` nonterminals are used for the syntax of conjunctive cases. Note that we do not need to write explicitly the semicolon between the boolean and the rest of the case, because it is already part of the syntax of `Exp`.

```
syntax Case ::= "|" Exp "=>" Exp [strict(2)]
```

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

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

```
syntax Type ::= Id
             | Id "(" Types ")"
syntax Types ::= List{Type,","}
```

Types can be basic types, function types, or user-defined parametric types. In the dynamic semantics we are going to simply ignore types, but we will parse them and check that they are well-formed. Function types are right-associative, as usual. Parametric types are parsed tentatively with the arrow construct for function types, not the symbol `->` as a convention for function types.

```
syntax Type ::= Id
             | "Int" | "Bool"
             | Type "->" Type [right]
             | "(" Type ")" [bracket]
             | Id "(" Types ")" [prefer]
             | "(" Types ")" "->" Type [prefer]
```

Conjunctive cases are used in `case` statements, as well as in datatype declarations. Each conjunctive case starts with a capital identifier, which is the name of the data constructor, and it can be followed by an arbitrary number of types enclosed in parentheses. Parentheses are not needed when the data constructor takes no arguments.

```
syntax TypeCases ::= List{TypeCase,"|"}
syntax TypeCase ::= Id
                  | Id "(" Types ")"
```

The dynamic semantics ignore all the type declarations.

Additional Priorities

```
context Exp "+" Exp
```

Desugaring macros

We designate the K tool conventional operations to functions resolving over list operations. In order to do that we need some new variables. Note that we cannot use regular variables, because they will be captured by the program variables. Instead, we start creating with the character `!`, so we can easily recognize them when we debug or trace the semantics.

```
macro !X = !X
macro !Y = !Y
macro !Z = !Z
macro !Xs = !Xs
macro !Ys = !Ys
macro !Zs = !Zs
macro !Ks = !Ks
macro !Vs = !Vs
macro !Ls = !Ls
macro !Ts = !Ts
macro !Ps = !Ps
macro !Es = !Es
macro !Ds = !Ds
macro !Cs = !Cs
macro !Bs = !Bs
macro !Ss = !Ss
macro !Ms = !Ms
macro !Rs = !Rs
macro !Ns = !Ns
macro !Fs = !Fs
macro !Gs = !Gs
macro !Hs = !Hs
macro !Is = !Is
macro !Js = !Js
macro !Ws = !Ws
macro !Ys = !Ys
macro !Zs = !Zs
```

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

```
syntax Exp ::= Id "[" Types "]" [strict(2)]
```

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

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

```
syntax Type ::= Id
             | Id "(" Types ")"
             | Type "->" Type [right]
             | "(" Type ")" [bracket]
             | Id "(" Types ")" [prefer]
             | "(" Types ")" "->" Type [prefer]
syntax Types ::= List{Type,","}
```

Types can be basic types, function types, or user-defined parametric types. In the dynamic semantics we are going to simply ignore types, but we will parse them and check that they are well-formed. Function types are right-associative, as usual. Parametric types are parsed tentatively with the arrow construct for function types, not the symbol `->` as a convention for function types.

```
syntax TypeName ::= Id
                  | Id "(" Ids ")" [prefer]
syntax Ids ::= List{Id,","}
```

Conjunctive cases are used in `case` statements, as well as in datatype declarations. Each conjunctive case starts with a capital identifier, which is the name of the data constructor, and it can be followed by an arbitrary number of types enclosed in parentheses. Parentheses are not needed when the data constructor takes no arguments.

```
syntax TypeCases ::= List{TypeCase,"|"}
syntax TypeCase ::= Id
                  | Id "(" Types ")"
```

The dynamic semantics ignore all the type declarations.

Additional Priorities

```
context Exp "+" Exp
```

Desugaring macros

We designate the K tool conventional operations to functions resolving over list operations. In order to do that we need some new variables. Note that we cannot use regular variables, because they will be captured by the program variables. Instead, we start creating with the character `!`, so we can easily recognize them when we debug or trace the semantics.

```
macro !X = !X
macro !Y = !Y
macro !Z = !Z
macro !Xs = !Xs
macro !Ys = !Ys
macro !Zs = !Zs
```

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

```
syntax Exp ::= Id "[" Types "]" [strict(2)]
```

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

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

```
syntax Type ::= Id
             | Id "(" Types ")"
             | Type "->" Type [right]
             | "(" Type ")" [bracket]
             | Id "(" Types ")" [prefer]
             | "(" Types ")" "->" Type [prefer]
syntax Types ::= List{Type,","}
```

Types can be basic types, function types, or user-defined parametric types. In the dynamic semantics we are going to simply ignore types, but we will parse them and check that they are well-formed. Function types are right-associative, as usual. Parametric types are parsed tentatively with the arrow construct for function types, not the symbol `->` as a convention for function types.

```
syntax TypeName ::= Id
                  | Id "(" Ids ")" [prefer]
syntax Ids ::= List{Id,","}
```

Conjunctive cases are used in `case` statements, as well as in datatype declarations. Each conjunctive case starts with a capital identifier, which is the name of the data constructor, and it can be followed by an arbitrary number of types enclosed in parentheses. Parentheses are not needed when the data constructor takes no arguments.

```
syntax TypeCases ::= List{TypeCase,"|"}
syntax TypeCase ::= Id
                  | Id "(" Types ")"
```

The dynamic semantics ignore all the type declarations.

Additional Priorities

```
context Exp "+" Exp
```

Desugaring macros

We designate the K tool conventional operations to functions resolving over list operations. In order to do that we need some new variables. Note that we cannot use regular variables, because they will be captured by the program variables. Instead, we start creating with the character `!`, so we can easily recognize them when we debug or trace the semantics.

```
macro !X = !X
macro !Y = !Y
macro !Z = !Z
macro !Xs = !Xs
macro !Ys = !Ys
macro !Zs = !Zs
```

Semantics

The semantics below is environment-based. A substitution-based definition of FUN is also available, but does drop the K extension as explained above.

MODULE FUN-SETTINGS

Configuration

The K, store, and return cells are standard (see, for example, the definition of LAMBDA1++ or IMP++ in the first part of the K tutorial).

CONFIGURATION

Values and results

We take values to be integers, booleans and strings as values here, but will add more values later.

```
syntax Val ::= Int
             | Bool
             | String

syntax Val ::= Exps "(" ")"

syntax KResult ::= Val
```

Lookup

Arithmetic expressions

Conditional

Lists

We have already declared the syntactic list of expressions when, as we can assume that all the elements that appear as 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 operator builds lists of values when applied to values. Note that we do not have to say that the list operator is strict in its arguments, because the list arguments are lists and lists only build (or, when applicable, destruct) values. Also, we do not have to say that lists build lists of values specifically, it would be a characteristic value that requires the second, list argument.

```
syntax Val ::= List
               [nil]
syntax List ::= List List [assoc]
```

Data Constructors

Constructors take values as arguments and produce other values:

```
syntax Val ::= Constructor
             | Constructor(Vals)
```

Functions and Closures

Like in the environment-based semantics of LAMBDA1++ in the first part of the K tutorial, functions evaluate to closures. Closures are pairs consisting of a function and an environment. The environment is necessary to remember the variables that appear free in the function body (see static scoping in FUN).

```
syntax Val ::= Closure [Map.Closure]
```


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 the LAMBDA1++ definition from the first K tutorial.

To set up the pattern matching mechanism we need to specify which terms are to variables (for pattern matching, substitution and free variables). Variables are identifiers, but we also allow the list of identifiers to be variables, as we will use it for lists. Note that identifiers as *data* variables (in pattern matching, note that the klabel `Cons(Name,Identifiers)` can not variables after constructor data values):

```
syntax Variable ::= Name
```

We distinguish two cases when the closure is applied. If only the first pattern matches, then we pull the first case's body in the computation and we record the environment of the closure. If the first pattern does not match, then we put the first case in the `Cases` construct, and we record the environment of the closure. We also have to record the environment of the closure, as we will see later.

```
syntax Cases ::= Case [klabel(.Cases)]
               | Case Cases [klabel(_Cases_)]
```


Let and Letrec

*To highlight the similarities and differences between `let` and `letrec`, we prefer to give them almost symmetric kompat definitions. Both `let` and `letrec` have a list of declarations and a body. Both `let` and `letrec` are strict in the body. Additionally, `letrec` is strict in the declarations. In `let`, the declarations are not strict, because we want to be able to use them as arguments. Also, it binds the variables in `X1 ; X2 ; ...` to the corresponding values in `V1 ; V2 ; ...` (the corresponding variables and values are matched by position). In `letrec`, the declarations are strict, because we want to be able to use them recursively. The second difference is that the expression `V` is now one of the elements of the variables `X`, in the declarations