FUN — Untyped

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

Abstract

This is the 𝕂 semantic definition of the untyped FUN language. FUN is intended to be 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 evaluate to closures, which are first class values in the language. To make it more interesting and to highlight some of 𝕂's strengths, FUN includes the following features in addition to the conventional functional constructs encountered in similar languages used as teaching material:

- Functions can take multiple arguments in two different ways. First, they can take space-separated arguments whose semantics is given by currying. Second, they can take comma-separated tuple arguments, whose semantics is given directly, not via currying. For example, FUN allows function declarations/invocations of the form "f (a,b) c (d,e)".
- Similarly, we allow `let` and `letrec` binders which work with lists of variables and expressions, and we give their semantics directly, without desugaring them to one-argument variants. We also allow the usual syntactic sugar for declaring-and-binding functions with "`let f (a,b) c (d,e) = ...`".
- We include a `callcc` construct, for two reasons: first, several functional languages support this construct; second, some semantic frameworks have a hard time defining it.
- Finally, we include mutables by means of referencing, dereferencing and assignments. We include these for the same reasons as above: there are functional 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 in a dedicated module between the syntax and the semantics, and we only give semantics to the core constructs.

Note: For a quick introduction to the 𝕂 prototype, you are referred to the README file at the root of this k-framework distribution. If you are interested in reading more about 𝕂, please check the following paper:

Grigore Roşu, Traian-Florin Şerbănuţă: An overview of the K semantic framework.
Journal of Logic and Algebraic Programming, 79(6): 397-434 (2010)

MODULE FUN-UNTYPED-SYNTAX
SYNTAX Exp ::= #Int
| #Bool
| #String
| #Id
| Exp `+` Exp [strict]
| Exp `-` Exp [strict]
| Exp `*` Exp [strict]
| Exp `/` Exp [strict]
| Exp `%` Exp [strict]
| `-` Exp [strict]
| Exp `<` Exp [strict]
| Exp `<=` Exp [strict]
| Exp `>` Exp [strict]
| Exp `>=` Exp [strict]
| Exp `==` Exp [strict]
| Exp `!=` Exp [strict]
| Exp `and` Exp [strict]
| Exp `or` Exp [strict]
| `not` Exp [strict]
| `fun` Ids `->` Exp
| Exp $Exps$ [strict]
| `let` Ids `=` $Exps$ `in` Exp [strict(2)]
| `letrec` Ids `=` $Exps$ `in` Exp
| `if` Exp `then` Exp `else` Exp [strict(1)]
| `[` $Exps$ `]` [hybrid strict]
| `car` Exp [strict]
| `cdr` Exp [strict]
| `null?` Exp [strict]
| `cons` Exp Exp [strict]
| `ref` Exp [strict]
| `&` #Id
| `*` Exp [strict]
| Exp `:=` Exp [strict]
| Exp `;` Exp [seqstrict]
| `callcc` Exp [strict]
SYNTAX Ids ::= List{#Id,","}
SYNTAX $Exps$ ::= List{Exp,","}
END MODULE

MODULE FUN-UNTYPED
IMPORTS FUN-UNTYPED-SYNTAX
SYNTAX $Vals$ ::= List{Val,","}
SYNTAX Val ::= #Int
| #Bool
SYNTAX Exp ::= Val
SYNTAX $KResult$ ::= Val

Configuration

CONFIGURATION:

T: k: $PGM · env: • · store: • · nextLoc: 0

Lookup

RULE k: $X \Rightarrow V$ · env: $X \mapsto L$ · store: $L \mapsto V$

Arithmetic expressions

RULE I_1 `+` $I_2 \Rightarrow I_1 +_{Int} I_2$

RULE I_1 `-` $I_2 \Rightarrow I_1 -_{Int} I_2$

RULE I_1 `*` $I_2 \Rightarrow I_1 *_{Int} I_2$

RULE I_1 `/` $I_2 \Rightarrow I_1 \div_{Int} I_2$ when $I_2 \neq_{Bool} 0$

RULE I_1 `%` $I_2 \Rightarrow I_1 \%_{Int} I_2$ when $I_2 \neq_{Bool} 0$

RULE `-` $I \Rightarrow -_{Int} I$

RULE I_1 `<` $I_2 \Rightarrow I_1 <_{Int} I_2$

RULE I_1 `<=` $I_2 \Rightarrow I_1 \leq_{Int} I_2$

RULE I_1 `>` $I_2 \Rightarrow I_1 >_{Int} I_2$

RULE I_1 `>=` $I_2 \Rightarrow I_1 \geq_{Int} I_2$

RULE V_1 `==` $V_2 \Rightarrow V_1 =_{Bool} V_2$

RULE V_1 `!=` $V_2 \Rightarrow V_1 \neq_{Bool} V_2$

RULE T_1 `and` $T_2 \Rightarrow T_1 \wedge_{Bool} T_2$

RULE T_1 `or` $T_2 \Rightarrow T_1 \vee_{Bool} T_2$

RULE `not` $T \Rightarrow \neg_{Bool} T$

Functions and Closures

SYNTAX Val ::= `closure(` Map `,` Ids `,` Exp `)`

RULE k: `fun` Xs `->` E ⇒ `closure(` ρ `,` Xs `,` E `)` · env: ρ

RULE k: `closure(` ρ `,` Xs `,` E `)` Vs ⇒ `bindTo(` Xs `,` Vs `)` $\curvearrowright E \curvearrowright$ `env(` ρ' `)` · env: $\rho' \Rightarrow \rho$

Let

RULE k: `let` Xs `=` Vs `in` E ⇒ `bindTo(` Xs `,` Vs `)` $\curvearrowright E \curvearrowright$ `env(` ρ `)` · env: ρ

Letrec

RULE k: `letrec` Xs `=` Es `in` E ⇒ `bind(` Xs `)` $\curvearrowright$ `assignTo(` Xs `,` Es `)` $\curvearrowright E \curvearrowright$ `env(` ρ `)` · env: ρ

Callcc

SYNTAX Val ::= `cc(` Map `,` K `)`

RULE k: `callcc` V ⇒ V `cc(` ρ `,` K `)` $\curvearrowright K$ · env: ρ

RULE k: `cc(` ρ `,` K `)` $V \curvearrowright$ – ⇒ $V \curvearrowright K$ · env: – ⇒ ρ

Conditional

RULE `if true then` E `else` – $\Rightarrow E$

RULE `if false then` – `else` $E \Rightarrow E$

Lists

RULE `car [` V `,` – `]` $\Rightarrow V$

RULE `cdr [` – `,` Vs `]` $\Rightarrow$ `[` Vs `]`

RULE `null? []` $\Rightarrow$ `true`

RULE `null? [` – `,` – `]` $\Rightarrow$ `false`

RULE `cons` V `[` Vs `]` $\Rightarrow$ `[` V `,` Vs `]`

References

RULE k: `ref` V ⇒ L · store: • ⇒ $L \mapsto V$ · nextLoc: $L \Rightarrow s_{Nat}\ L$

RULE k: `&` X ⇒ L · env: $X \mapsto L$

RULE k: `*` L ⇒ V · store: $L \mapsto V$

RULE k: L `:=` V ⇒ V · store: $L \mapsto$ – ⇒ V

RULE V_1 `;` $V_2 \Rightarrow V_2$

Auxilliary operations

Environment recovery

SYNTAX K ::= `env(` Map `)`

RULE `env(` – `)` ⇒ • $\curvearrowright$ `env(` – `)` [structural]

RULE k: `env(` ρ `)` ⇒ • · env: – ⇒ ρ [structural]

RULE k: $V \curvearrowright$ `env(` ρ `)` ⇒ • · env: – ⇒ ρ [structural]

bindTo and bind

SYNTAX K ::= `bindTo(` Ids `,` $Vals$ `)`
| `bind(` Ids `)`

RULE k: `bindTo( • , • )` ⇒ • [structural]

RULE k: `bindTo(` X `,` Xs ⇒ Xs `,` V `,` Vs ⇒ Vs `)` · env: $\rho \Rightarrow \rho[N / X]$ · store: • ⇒ $N \mapsto V$ · nextLoc: $N \Rightarrow s_{Nat}\ N$ [structural]

RULE k: `bind( • )` ⇒ • [structural]

RULE k: `bind(` X `,` Xs ⇒ Xs `)` · env: $\rho \Rightarrow \rho[N / X]$ · nextLoc: $N \Rightarrow s_{Nat}\ N$ [structural]

assignTo

SYNTAX K ::= `assignTo(` Ids `,` $Exps$ `)` [strict(2)]

RULE k: `assignTo( • , • )` ⇒ • [structural]

RULE k: `assignTo(` X `,` Xs ⇒ Xs `,` V `,` Vs ⇒ Vs `)` · env: $X \mapsto L$ · store: • ⇒ $L \mapsto V$ [structural]

END MODULE