

FUN — Untyped — Substitution

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

Abstract

This is the substitution-based definition of FUN. For additional explanations regarding the semantics of the various FUN constructs, the reader should consult the environment-based definition of FUN.

Syntax

MODULE FUN-UNTYPED-SYNTAX

The Syntactic Constructs

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

SYNTAX $Names ::= List\{Name, ", "\}$

SYNTAX $Exp ::= Int$
 $\quad \quad \quad | Bool$
 $\quad \quad \quad | String$
 $\quad \quad \quad | Name$
 $\quad \quad \quad | (Exp) \text{ [bracket]}$

SYNTAX $Exps ::= List\{Exp, ", "\} \text{ [strict]}$

SYNTAX $Exp ::= Exp * Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp / Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp \% Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp + Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp - Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp * Exp \text{ [strict, prefer, arith]}$
 $\quad \quad \quad | Exp < Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp <= Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp > Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp >= Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp == Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp != Exp \text{ [strict, arith]}$
 $\quad \quad \quad | ! Exp \text{ [strict, arith]}$
 $\quad \quad \quad | Exp \&\& Exp \text{ [strict(1), arith]}$
 $\quad \quad \quad | Exp || Exp \text{ [strict(1), arith]}$

SYNTAX $Exp ::= \text{if } Exp \text{ then } Exp \text{ else } Exp \text{ [strict(1)]}$

SYNTAX $Exp ::= [Exps] \text{ [strict]}$
 $\quad \quad \quad | \text{cons}$
 $\quad \quad \quad | \text{head}$
 $\quad \quad \quad | \text{tail}$
 $\quad \quad \quad | \text{null?}$
 $\quad \quad \quad | [Exps \mid Exp]$

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

SYNTAX $Exp ::= ConstructorName$
 $\quad \quad \quad | ConstructorName(Exps) \text{ [prefer, strict(2)]}$

SYNTAX $Exp ::= \text{fun } Cases$
 $\quad \quad \quad | Exp \text{ Exp [strict]}$

SYNTAX $Case ::= Exp \rightarrow Exp \text{ [binder]}$

SYNTAX $Cases ::= List\{Case, ", "\}$

SYNTAX $Exp ::= \text{let } Bindings \text{ in } Exp$
 $\quad \quad \quad | \text{letrec } Bindings \text{ in } Exp \text{ [prefer]}$

SYNTAX $Binding ::= Exp = Exp$

SYNTAX $Bindings ::= List\{Binding, "and" \}$

SYNTAX $Exp ::= \text{ref}$
 $\quad \quad \quad | \& Name$
 $\quad \quad \quad | @ Exp \text{ [strict]}$
 $\quad \quad \quad | Exp := Exp \text{ [strict]}$
 $\quad \quad \quad | Exp ; Exp \text{ [strict(1)]}$

SYNTAX $Exp ::= \text{callcc}$
 $\quad \quad \quad | \text{try } Exp \text{ catch } (Name)Exp$

SYNTAX $Name ::= \text{throw}$

SYNTAX $Exp ::= \text{datatype } Type = TypeCases \text{ Exp}$

SYNTAX $TypeVar ::= Token\{[_][_a - z][_a - zA - Z0 - 9]^*\} \text{ [notInRules]}$

SYNTAX $TypeVars ::= List\{TypeVar, ", "\}$

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

SYNTAX $Type ::= \text{int}$
 $\quad \quad \quad | \text{bool}$
 $\quad \quad \quad | \text{string}$
 $\quad \quad \quad | Type \rightarrow Type$
 $\quad \quad \quad | (Type) \text{ [bracket]}$
 $\quad \quad \quad | TypeVar$
 $\quad \quad \quad | TypeName \text{ [klabel('TypeName), onlyLabel]}$
 $\quad \quad \quad | Type \text{ TypeName [klabel('Type-TypeName), onlyLabel]}$
 $\quad \quad \quad | (Types)TypeName \text{ [prefer]}$

SYNTAX $Types ::= List\{Type, ", "\}$

SYNTAX $TypeCase ::= ConstructorName$
 $\quad \quad \quad | ConstructorName(Types)$

SYNTAX $TypeCases ::= List\{TypeCase, "\}"$

Additional Priorities

Desugaring macros

RULE $\frac{P1 \ P2 \rightarrow E}{P1 \rightarrow \text{fun } P2 \rightarrow E}$ [macro]

RULE $\frac{F \ P = E}{F = \text{fun } P \rightarrow E}$ [macro]

RULE $\frac{[E, Es \mid T]}{[E \mid [Es \mid T]]}$ requires $Es \neq_K *_{\text{copy}}$ [macro]

RULE $\frac{'TypeName(Tn)}{(*_{TypeVar})Tn}$ [macro]

RULE $\frac{'Type - TypeName(T, Tn)}{(T)Tn}$ [macro]

SYNTAX $Name ::= \$h$
 $\quad \quad \quad | \$t$

RULE $\frac{\text{head}}{\text{fun } [\$h \mid \$t] \rightarrow \$h}$ [macro]

RULE $\frac{\text{tail}}{\text{fun } [\$h \mid \$t] \rightarrow \$t}$ [macro]

RULE $\frac{\text{null?}}{\text{fun } [*_{\text{copy}}] \rightarrow \text{true} \mid [\$h \mid \$t] \rightarrow \text{false}}$ [macro]

SYNTAX $Name ::= \$k$
 $\quad \quad \quad | \$v$

RULE $\frac{\text{try } E \text{ catch } (X)E'}{\text{callcc } (\text{fun } \$k \rightarrow (\text{fun } \text{throw} \rightarrow E) \ (\text{fun } X \rightarrow \$k \ E'))}$ [macro]

RULE $\frac{\text{datatype } T = TCs \ E}{E}$ [macro]

mu needed for letrec, but we put it here so we can also write programs with mu in them, which is particularly useful for testing.

SYNTAX $Exp ::= \text{mu } Case$

END MODULE

Semantics

MODULE FUN-UNTYPED



Both Name and functions are values now:

SYNTAX $Val ::= Int$
 $\quad \quad \quad | Bool$
 $\quad \quad \quad | String$
 $\quad \quad \quad | Name$

SYNTAX $Vals ::= List\{Val, ", "\}$

SYNTAX $Exp ::= Val$

SYNTAX $KResult ::= Val$

RULE $\frac{I1 * I2}{I1 *_{\text{fst}} I2}$ requires $I2 \neq_K 0$

RULE $\frac{I1 / I2}{I1 \div_{\text{fst}} I2}$ requires $I2 \neq_K 0$

RULE $\frac{I1 \% I2}{I1 \%_{\text{fst}} I2}$ requires $I2 \neq_K 0$

RULE $\frac{I1 + I2}{I1 +_{\text{fst}} I2}$

RULE $\frac{S1 + S2}{S1 +_{\text{String}} S2}$

RULE $\frac{I1 - I2}{I1 -_{\text{fst}} I2}$

RULE $\frac{I1 < I2}{I1 <_{\text{fst}} I2}$

RULE $\frac{I1 \Leftarrow I2}{I1 \Leftarrow_{\text{fst}} I2}$

RULE $\frac{I1 > I2}{I1 >_{\text{fst}} I2}$

RULE $\frac{I1 \geq I2}{I1 \geq_{\text{fst}} I2}$

RULE $\frac{V1 == V2}{V1 ==_K V2}$

RULE $\frac{V1 != V2}{V1 \neq_K V2}$

RULE $\frac{! T}{\neg Bool(T)}$

RULE $\frac{\text{true } \&\& E}{E}$

RULE $\frac{\text{false } \&\& \text{---}}{\text{false}}$

RULE $\frac{\text{true } || \text{---}}{\text{true}}$

RULE $\frac{\text{false } || E}{E}$

RULE $\frac{\text{if true then } E \text{ else ---}}{E}$

RULE $\frac{\text{if false then --- else } E}{E}$

SYNTAX $Val ::= \text{cons}$
 $\quad \quad \quad | [Vals]$

RULE $\frac{\text{isVal}(\text{cons } V)}{\text{true}}$

RULE $\frac{\text{cons } V \ [Vs]}{[V, Vs]}$

SYNTAX $Val ::= ConstructorName$
 $\quad \quad \quad | ConstructorName(Vals)$

SYNTAX $Val ::= \text{fun } Cases$

SYNTAX $Variable ::= Name$

RULE $\frac{(\text{fun } P \rightarrow E \mid \text{---}) \ V}{E[\text{getMatching}(P, V)]}$ requires $\text{isMatching}(P, V)$

RULE $\frac{(\text{fun } P \rightarrow \text{---} \mid Cs) \ V}{Cs}$ requires $\neg_{Bool} \text{isMatching}(P, V)$

RULE $\frac{\text{decomposeMatching}([H \mid T], [V, Vs])}{H, T \ V, [Vs]}$

We can reduce multiple bindings to one list binding, and then apply the usual desugaring of let into function application. It is important that the rule below is a macro, so let is eliminated immediately, otherwise it may interfere in ugly ways with substitution.

RULE $\frac{\text{let } Bs \text{ in } E}{((\text{fun } [names \ (Bs)] \rightarrow E) \mid \text{exps } (Bs))}$ [macro]

We only give the semantics of one-binding letrec. Multipe bindings are left as an exercise.

RULE $\frac{\text{mu } X \rightarrow E}{E[(\text{mu } X \rightarrow E) / X]}$

RULE $\frac{\text{letrec } F = E \text{ in } E'}{\text{let } F = (\text{mu } F \rightarrow E) \text{ in } E'}$ [macro]

We cannot have & anymore, but we can give direct semantics to ref. We also have to declare ref to be a value, so that we will never heat on it.

SYNTAX $Val ::= \text{ref}$

RULE $\frac{\text{ref } V}{L}$ requires $\text{fresh } (L)$

RULE $\frac{\text{store } L \mapsto V}{@ L}{V}$

RULE $\frac{\text{store } L \mapsto \text{---}}{L := V}{V}$

RULE $\frac{V ; E}{E}$

SYNTAX $Val ::= \text{callcc}$
 $\quad \quad \quad | \text{cc } (K)$

RULE $\frac{\text{callcc } V \curvearrowright K}{V \curvearrowright \text{cc } (K)}$

RULE $\frac{\text{cc } (K) \ V \curvearrowright \text{---}}{V \curvearrowright K}$

Auxiliary getters

SYNTAX $Names ::= \text{names } (Bindings) \text{ [function]}$

RULE $\frac{\text{names } (*_{Bindings})}{*_{Names}}$

RULE $\frac{\text{names } (X = \text{---and } Bs)}{X, \text{names } (Bs)}$

SYNTAX $Exps ::= \text{exps } (Bindings) \text{ [function]}$

RULE $\frac{\text{exps } (*_{Bindings})}{*_{Exps}}$

RULE $\frac{\text{exps } (\text{---Eand } Bs)}{E, \text{exps } (Bs)}$

END MODULE