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 emvironment-based definition of FUN.

Syntax

MODULE FUN-UNTYPED-SYNTAX

The Syntactic Constructs

SYNTAX *Name* ::= [notInRules(notInRules())]

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

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

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

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

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

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

SYNTAX *ConstructorName* ::= [notInRules(notInRules())]

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

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

SYNTAX *Case* ::= *Exp* -> *Exp* [binder(binder())]

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

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

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

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

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

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

SYNTAX *Name* ::= `throw`

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

SYNTAX *TypeVar* ::= [notInRules(notInRules())]

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

SYNTAX *TypeName* ::= [notInRules(notInRules())]

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

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

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

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

Additional Priorities

END MODULE

MODULE FUN-UNTYPED-MACROS

Desugaring macros

RULE $\dfrac{P1\ \ P2\ \texttt{->}\ E}{P1\ \texttt{->}\ \texttt{fun}\ P2\ \texttt{->}\ E}$ [macro(macro())]

RULE $\dfrac{F\ \ P\ \texttt{=}\ E}{F\ \texttt{=}\ \texttt{fun}\ P\ \texttt{->}\ E}$ [macro(macro())]

RULE $\dfrac{[E{:}Exp\texttt{,}\ Es\ \texttt{|}\ T]}{[E\ \texttt{|}\ [Es\ \texttt{|}\ T]]}$ requires $Es \neq_K \bullet_{Exps}$ [macro(macro())]

RULE $\dfrac{'TypeName(Tn{:}TypeName)}{(\bullet_{TypeVars})\,Tn}$ [macro(macro())]

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

SYNTAX *Name* ::= `$h`
| `$t`

RULE $\dfrac{\texttt{head}}{\texttt{fun [\$h | \$t] -> \$h}}$ [macro(macro())]

RULE $\dfrac{\texttt{tail}}{\texttt{fun [\$h | \$t] -> \$t}}$ [macro(macro())]

RULE $\dfrac{\texttt{null?}}{\texttt{fun}\ [\bullet_{Exps}]\ \texttt{-> true | [\$h | \$t] -> false}}$ [macro(macro())]

SYNTAX *Name* ::= `$k`
| `$v`

RULE $\dfrac{\texttt{try}\ E\ \texttt{catch}\ (X)E'}{\texttt{callcc}\ (\ \texttt{fun \$k ->}\ (\ \texttt{fun throw ->}\ E)\ \ (\ \texttt{fun}\ X\ \texttt{-> \$k}\ \ E'))}$ [macro(macro())]

RULE $\dfrac{\texttt{datatype}\ T\ \texttt{=}\ TCs\ \ E}{E}$ [macro(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* ::= `mu` *Case*

END MODULE

Semantics

MODULE FUN-UNTYPED

CONFIGURATION:

$\langle\, \langle \$PGM{:}Exp \rangle_{\texttt{k}}\ \ \langle \texttt{.Map} \rangle_{\texttt{store}} \,\rangle_{\texttt{T}}$

Both Name and functions are values now:

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

SYNTAX *Vals* ::= $List\{Val, \text{“,”}\}$

SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

RULE $\dfrac{I1{:}Int\ \texttt{*}\ I2{:}Int}{I1 *_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{/}\ I2{:}Int}{I1 \div_{Int} I2}$ requires $I2 \neq_K \texttt{0}$

RULE $\dfrac{I1{:}Int\ \texttt{\%}\ I2{:}Int}{I1 \,\%_{Int}\, I2}$ requires $I2 \neq_K \texttt{0}$

RULE $\dfrac{I1{:}Int\ \texttt{+}\ I2{:}Int}{I1 +_{Int} I2}$

RULE $\dfrac{S1{:}String\ \texttt{\^{}}\ S2{:}String}{S1 +_{String} S2}$

RULE $\dfrac{I1{:}Int\ \texttt{-}\ I2{:}Int}{I1 -_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{<}\ I2{:}Int}{I1 <_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{<=}\ I2{:}Int}{I1 \leq_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{>}\ I2{:}Int}{I1 >_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{>=}\ I2{:}Int}{I1 \geq_{Int} I2}$

RULE $\dfrac{V1{:}Val\ \texttt{==}\ V2{:}Val}{V1 =_K V2}$

RULE $\dfrac{V1{:}Val\ \texttt{!=}\ V2{:}Val}{V1 \neq_K V2}$

RULE $\dfrac{\texttt{!}\ T{:}Bool}{\neg_{Bool}(T)}$

RULE $\dfrac{\texttt{true \&\&}\ E}{E}$

RULE $\dfrac{\texttt{false \&\&}\ \text{—}}{\texttt{false}}$

RULE $\dfrac{\texttt{true ||}\ \text{—}}{\texttt{true}}$

RULE $\dfrac{\texttt{false ||}\ E}{E}$

RULE $\dfrac{\texttt{if true then}\ E\ \texttt{else}\ \text{—}}{E}$

RULE $\dfrac{\texttt{if false then}\ \text{—}\ \texttt{else}\ E}{E}$

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

RULE $\dfrac{isVal(\ \texttt{cons}\ \ V{:}Val)}{\texttt{true}}$

RULE $\dfrac{\texttt{cons}\ \ V{:}Val\ \ [Vs{:}Vals]}{[V\texttt{,}\ Vs]}$

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

SYNTAX *Val* ::= `fun` *Cases*

SYNTAX *Variable* ::= *Name*

SYNTAX *Name* ::= `freshName` (*Int*) [freshGenerator(freshGenerator()), function(function()), klabel(klabel('freshName))]

RULE $\dfrac{\texttt{freshName}\ (I{:}Int)}{\texttt{\#parseToken}\ (\texttt{"Name"}, \texttt{"\#"} +_{String} \texttt{Int2String}\ (I))}$

RULE $\dfrac{(\ \texttt{fun}\ P\ \texttt{->}\ E\,\texttt{|}\,\text{—})\ \ V{:}Val}{E[\ \texttt{getMatching}\ (P, V)]}$ requires $\texttt{isMatching}\ (P, V)$

RULE $\left(\texttt{fun}\ \dfrac{P\ \texttt{->}\ \text{—}\,\texttt{|}\ Cs{:}Cases}{Cs}\right)\ \ V{:}Val$ requires $\neg_{Bool}\ \texttt{isMatching}\ (P, V)$

RULE $\texttt{decomposeMatching}\ (\dfrac{[H{:}Exp\ \texttt{|}\ T{:}Exp]}{H\texttt{,}\ T}, \dfrac{[V{:}Val\texttt{,}\ Vs{:}Vals]}{V\texttt{,}\ [Vs]})$

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 $\dfrac{\texttt{let}\ Bs\ \texttt{in}\ E}{((\ \texttt{fun}\ [\ \texttt{names}\ (Bs)]\ \texttt{->}\ E)\ \ [\ \texttt{exps}\ (Bs)])}$ [macro(macro())]

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

RULE $\dfrac{\texttt{mu}\ X{:}Name\ \texttt{->}\ E}{E[(\ \texttt{mu}\ X\ \texttt{->}\ E)\ \texttt{/}\ X]}$

RULE $\dfrac{\texttt{letrec}\ F{:}Name\ \texttt{=}\ E\ \texttt{in}\ E'}{\texttt{let}\ F\ \texttt{=}\ (\ \texttt{mu}\ F\ \texttt{->}\ E)\ \texttt{in}\ E'}$ [macro(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* ::= `ref`

RULE $\left\langle \dfrac{\texttt{ref}\ \ V{:}Val}{L{:}Int} \cdots \right\rangle_{\texttt{k}}\ \ \left\langle \cdots \dfrac{\texttt{.Map}}{L \mapsto V} \cdots \right\rangle_{\texttt{store}}$

RULE $\left\langle \dfrac{\texttt{@}\ L{:}Int}{V{:}Val} \cdots \right\rangle_{\texttt{k}}\ \ \left\langle \cdots L \mapsto V \cdots \right\rangle_{\texttt{store}}$

RULE $\left\langle \dfrac{L{:}Int\ \texttt{:=}\ V{:}Val}{V} \cdots \right\rangle_{\texttt{k}}\ \ \left\langle \cdots L \mapsto \dfrac{\text{—}}{V} \cdots \right\rangle_{\texttt{store}}$

RULE $\dfrac{V{:}Val\ \texttt{;}\ E}{E}$

SYNTAX *Val* ::= `callcc`
| `cc` (*K*) [klabel(klabel('cc))]

RULE $\left\langle \dfrac{\texttt{callcc}\ \ V{:}Val}{V\ \ \texttt{cc}\ (K)} \curvearrowright K \right\rangle_{\texttt{k}}$

RULE $\left\langle \dfrac{\texttt{cc}\ (K)\ \ V{:}Val \curvearrowright \text{—}}{V \curvearrowright K} \right\rangle_{\texttt{k}}$

Auxiliary getters

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

RULE $\dfrac{\texttt{names}\ (\bullet_{Bindings})}{\bullet_{Names}}$

RULE $\dfrac{\texttt{names}\ (X{:}Name\ \texttt{=}\ \text{—}\texttt{and}\ Bs)}{X\texttt{,}\ \texttt{names}\ (Bs)}$

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

RULE $\dfrac{\texttt{exps}\ (\bullet_{Bindings})}{\bullet_{Exps}}$

RULE $\dfrac{\texttt{exps}\ (\text{—}{:}\mathrm{Name}\ \texttt{=}\ E\texttt{and}\ Bs)}{E\texttt{,}\ \texttt{exps}\ (Bs)}$

END MODULE