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, autoReject, notInRules]

SYNTAX *Names* ::= *List*{*Name*, ","}

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

SYNTAX *Exps* ::= *List*{*Exp*, ","} [strict]

SYNTAX *Exp* ::= *Exp* + *Exp* [strict, arith]
| *Exp* / *Exp* [strict, arith]
| *Exp* % *Exp* [strict, arith]
| *Exp* * *Exp* [strict, arith]
| *Exp* ^ *Exp* [strict, arith]
| *Exp* - *Exp* [prefer, strict, 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]

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

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

SYNTAX *ConstructorName* ::= [token, autoReject, notInRules]

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

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

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

SYNTAX *Cases* ::= *List*{*Case*, "|"}

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

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

SYNTAX *Bindings* ::= *List*{*Binding*, "and"}

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

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

SYNTAX *Name* ::= `throw` [token]

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

SYNTAX *TypeVar* ::= [token, autoReject, notInRules]

SYNTAX *TypeVars* ::= *List*{*TypeVar*, ","}

SYNTAX *TypeName* ::= [token, autoReject, notInRules]

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

SYNTAX *Types* ::= *List*{*Type*, ","}

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

SYNTAX *TypeCases* ::= *List*{*TypeCase*, "|"}

Additional Priorities

END MODULE

MODULE FUN-UNTYPED-MACROS

Desugaring macros

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

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

RULE $\dfrac{E{:}Exp_1 \; Es \mid T}{\langle E \mid Es \mid T\rangle}$ requires $Es \neq_K \bullet_{List}$ [macro]

RULE $\dfrac{TypeName\{Tn{:}TypeName\}}{\langle \bullet_{List}\rangle\, Tn}$ [macro]

RULE $\dfrac{Type - TypeName\{T{:}Type, Tn{:}TypeName\}}{\langle T\rangle\, Tn}$ [macro]

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

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

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

RULE $\dfrac{}{\texttt{fun [}\bullet_{List}\texttt{] -> true | \$h | \$t -> false}}$ [macro]

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

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

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

END MODULE

Semantics

MODULE FUN-UNTYPED

CONFIGURATION:

Both Name and functions are values now:

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

SYNTAX *Vals* ::= *List*{*Val*, ","}

SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

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

RULE $\dfrac{I1{:}Int \;/\; I2{:}Int}{I1 /_{Int} I2}$ requires $I2 \neq_K 0$

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

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

RULE $\dfrac{S1{:}String \;\hat{}\; S2{:}String}{S1 +_{String} S2}$

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

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

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

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

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

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

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

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

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

RULE $\dfrac{\texttt{false \&\& }\_}{\texttt{false}}$

RULE $\dfrac{\texttt{true || }\_}{\texttt{true}}$

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

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

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

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

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

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

SYNTAX *Val* ::= *ConstructorName*
| *ConstructorName*{*Vals*}

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

SYNTAX *Variable* ::= *Name*

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

RULE $\dfrac{}{\texttt{freshName}\,(I{:}Int)}$

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

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

RULE $\dfrac{\texttt{fun }P\texttt{ -> }\_\texttt{ | }Cs{:}Cases}{Cs}\;\; V{:}Val$ requires $\neg_{Bool}$ `isMatching` (P, V)

RULE $\dfrac{\texttt{decomposeMatching}\,(\langle H{:}Exp \mid T{:}Exps\rangle, \langle V{:}Val, Vs{:}Vals\rangle)}{H, T \;\; V, \langle Vs\rangle}$

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

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

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

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

RULE 

RULE

RULE

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

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

RULE

RULE

Auxiliary getters

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

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

RULE $\dfrac{}{\bullet_{List}}$

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

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

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

RULE $\dfrac{\texttt{exps}\,(\_ = E\texttt{ and }Bs)}{E\texttt{ , exps }(Bs)}$

END MODULE