CINK

MODULE CINK-LANG-SYNTAX

SYNTAX *Decl* ::= int *Exps*
| void *Exps*
| bool *Exps*

SYNTAX *Val* ::= cout
| cin
| *Bool*
| *String*
| *Int*

SYNTAX *Exp* ::= *Id*
| *Val*
| (*Exp*)
| *Id*(*Exps*) [strict(1)]
| & *Exp*
| ∗ *Exp* [strict]
| ++ *Exp*
| *Exp* ∗ *Exp*
| *Exp* / *Exp*
| *Exp*%*Exp*
| *Exp* + *Exp*
| *Exp* - *Exp*
| *Exp* > *Exp*
| *Exp* ≤ *Exp*
| *Exp* == *Exp*
| ! *Exp* [strict]
| *Exp* && *Exp* [strict(1)]
| *Exp* || *Exp* [strict(1)]
| *Exp* << *Exp* [seqstrict]
| *Exp* >> *Exp*
| *Exp* = *Exp*
| endl

SYNTAX *#Id* ::= main

SYNTAX *Stmt* ::= *Exp* ; [strict]
| #include<iostream>
| using namespace std;
| *Decl* ;
| {}
| {*Stmts*}
| while (*Exp*)*Stmt*
| return *Exps* ;
| *Decl*(*Decls*){*Stmts*}
| if (*Exp*)*Stmt* else *Stmt* [strict(1)]
| if (*Exp*)*Stmt*
| std::thread *Id*(*Exps*) ;

SYNTAX *Pgm* ::= *Stmts*

SYNTAX *Stmts* ::= *Stmt*
| *Stmts* *Stmts*

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

SYNTAX *Decls* ::= *List*{*Decl*, “,”}

END MODULE

MODULE CINK-LANG-SEMANTICS

SYNTAX *LVal* ::= loc (*Int*)

SYNTAX *Val* ::= void
| undefined
| *LVal*
| λ *Decls* • *Stmts*

SYNTAX *Vals* ::= *List*{*Val*, “,”}

SYNTAX *KResult* ::= *Val*

SYNTAX *Exp* ::= lvalue (*Exp*)
| rvalue (*Exp*) [strict]

SYNTAX *K* ::= execute
| noname

SYNTAX *ListItem* ::= (*List*, *K*)
| [*Map*]

CONFIGURATION:

T: [threads: [thread: [name: noname | k: $PGM ↷ execute | env: •Map | bstack: •List | fstack: •List]] | genv: •Map | store: •Map | in: $IN | out: •List | nextLoc: 0]

RULE if (B)St ⇒ if (B)St else {} [anywhere]

RULE '_;(KL('_`,_(E, Es))) ⇒ '__('_;(KL('_`,_(E, •Exps))), '_;(KL(Es))) when isDeclLabel (KL) $\wedge_{Bool}$ Es =/=K •Exps [anywhere]

RULE '_;(KL('_`,_('_=_(E1, E2), •Exps))) ⇒ '__('_;(KL('_`,_(E1, •Exps))), '_;('_=_(getName (E1), E2))) when isDeclLabel (KL) $\wedge_{Bool}$ isAlias ('_=_(E1, E2)) =/=K true [anywhere]

RULE k: '_`(_`)`{_`}(Decl, Xl, Sts) ⇒ •K | env: •Map ⇒ getName (Decl) ↦ L | store: •Map ⇒ L ↦ λ Xl • Sts | nextLoc: L ⇒ L $+_{Int}$ 1 [structural]

CONTEXT KL('_`,_(& — = □ ⇒ lvalue (□), •Exps)) when isDeclLabel (KL)

RULE k: KL('_`,_(& X = loc (L), •Exps)) ⇒ void | env: Env ⇒ Env[L / X] when isDeclLabel (KL) [structural]

RULE k: KL('_`,_(E, •Exps)) ⇒ void | env: Env ⇒ Env[L / getName (E)] | store: •Map ⇒ L ↦ undefined | nextLoc: L ⇒ L $+_{Int}$ 1 when isDeclLabel (KL) $\wedge_{Bool}$ isAlias (E) =/=K true [structural]

RULE k: #include<iostream> ⇒ •K [structural]

RULE k: using namespace std; ⇒ •K [structural]

RULE k: execute ⇒ '_;('_`(_`)(main, •Vals)) | env: Env | genv: •Map ⇒ Env [structural]

CONTEXT □ ⇒ rvalue (□) + —

CONTEXT — + □ ⇒ rvalue (□)

RULE I1 + I2 ⇒ I1 $+_{Int}$ I2 [structural]

CONTEXT □ ⇒ rvalue (□) - —

CONTEXT — - □ ⇒ rvalue (□)

RULE I1 - I2 ⇒ I1 $-_{Int}$ I2 [structural]

CONTEXT □ ⇒ rvalue (□) ∗ —

CONTEXT — ∗ □ ⇒ rvalue (□)

RULE '_*_(I1, I2) ⇒ I1 $*_{Int}$ I2 [structural]

CONTEXT □ ⇒ rvalue (□) / —

CONTEXT — / □ ⇒ rvalue (□)

RULE I1 / I2 ⇒ I1 $\div_{Int}$ I2 when I2 =/=Int 0 [structural]

CONTEXT □ ⇒ rvalue (□) %—

CONTEXT —% □ ⇒ rvalue (□)

RULE I1%I2 ⇒ I1 $\%_{Int}$ I2 when I2 =/=Int 0 [structural]

CONTEXT □ ⇒ rvalue (□) > —

CONTEXT — > □ ⇒ rvalue (□)

RULE I1 > I2 ⇒ I1 $>_{Int}$ I2 [structural]

CONTEXT □ ⇒ rvalue (□) ≤ —

CONTEXT — ≤ □ ⇒ rvalue (□)

RULE I1 ≤ I2 ⇒ I1 $\leq_{Int}$ I2 [structural]

CONTEXT □ ⇒ rvalue (□) == —

CONTEXT — == □ ⇒ rvalue (□)

RULE I1 == I2 ⇒ I1 ==Int I2 [structural]

RULE true && B ⇒ B [structural]

RULE false && B ⇒ false [structural]

RULE true || B ⇒ true [structural]

RULE false || B ⇒ B [structural]

RULE ! false ⇒ true [structural]

RULE ! true ⇒ false [structural]

RULE endl ⇒ "\n" [structural]

RULE k: ∗ L ⇒ V | store: L ↦ V

CONTEXT & □ ⇒ lvalue (□)

RULE & loc (L) ⇒ L

CONTEXT ++ □ ⇒ lvalue (□)

RULE k: ++ loc (L) ⇒ loc (L) | store: L ↦ V ⇒ V $+_{Int}$ 1

CONTEXT lvalue (□) when isLValue (□) =/=K true

RULE lvalue (L) ⇒ loc (L) [structural]

RULE lvalue (loc (L)) ⇒ loc (L) [structural]

RULE k: rvalue (loc (L)) ⇒ V | store: L ↦ V

RULE rvalue (V) ⇒ V when 'isLVal(V) =/=K true

RULE k: lvalue (X) ⇒ loc (L) | env: X ↦ L [transition]

RULE k: lvalue (∗ X) ⇒ loc (V) | env: X ↦ L | store: L ↦ V [transition]

RULE k: X ⇒ V | env: X ↦ L | store: L ↦ V [transition]

CONTEXT □ ⇒ lvalue (□) = —

CONTEXT — = □ ⇒ rvalue (□)

RULE k: loc (L) = V ⇒ V | store: L ↦ — ⇒ V [transition]

RULE while (B)St ⇒ if (B){St while (B)St} else {} [structural]

RULE if (false)— else St ⇒ St

RULE if (true)St else — ⇒ St

RULE k: V ;↷ K ⇒ K when K =/=K •K

RULE k: {Sts} ⇒ Sts ↷popb | env: Env | bstack: •List ⇒ [Env] [structural]

RULE k: {Sts} ⇒ Sts ↷popb | env: Env | bstack: •List ⇒ [Env] [Env1] when Env =/=Map Env1 [structural]

RULE k: {Sts} ⇒ Sts | env: Env | bstack: [Env1] when Env ==Map Env1 [structural]

RULE {} ⇒ •K [structural]

RULE Sts Sts′ ⇒ Sts ↷ Sts′ [structural]

RULE k: cout << V ⇒ cout | out: •List ⇒ V " "

CONTEXT □ >> —

CONTEXT — >> □ ⇒ lvalue (□)

RULE k: cin >> loc (L) ⇒ cin | store: L ↦ — ⇒ I | in: I ⇒ •List

RULE k: '_`(_`)(λ Xl • Sts, Vl) ↷ K ⇒ bind Vl to Xl ;↷ Sts ↷return void, •Exps ; | genv: GEnv | env: Env ⇒ GEnv | bstack: Stack | fstack: •List ⇒ ([Env] Stack, K)

CONTEXT '_`(_`)(λ Xl • Sts, □ ⇒ evaluate □ following Xl ;)

SYNTAX *Exps* ::= evaluate *Exps* following *Decls* ;

CONTEXT evaluate □, — following int X , •Exps , — ;

CONTEXT evaluate □ ⇒ lvalue (□) , — following int & X , •Exps , — ;

RULE evaluate V , El following Dec, Xl ; ⇒ V , evaluate El following Xl ;

RULE evaluate •Exps following •Decls ; ⇒ •Vals

SYNTAX *K* ::= bind *Vals* to *Decls* ;

RULE k: bind V , Vs ⇒ Vs to int X , •Exps , Xl ; ⇒ Xl | env: Env ⇒ Env[L / X] | store: •Map ⇒ L ↦ V | nextLoc: L ⇒ L $+_{Int}$ 1

RULE k: bind loc (L), Vs ⇒ Vs to int & X , •Exps , Xl ; ⇒ Xl | env: Env ⇒ Env[L / X]

RULE bind •Vals to •Decls ; ⇒ •K [structural]

RULE return •Exps ; ⇒ return void, •Exps ; [transition]

CONTEXT return □, Es ;

RULE return V , Es ; ⇒ return Es ; when Es =/=K •Exps [transition]

RULE k: return V , •Exps ;↷ — ⇒ V ↷ K | env: — ⇒ Env | bstack: — ⇒ Stack | fstack: ([Env] Stack, K) ⇒ •List [transition]

RULE thread: [k: 'std::thread_`(_`);(T, '_`,_(F, El)) ⇒ •K | env: Env ] •Bag ⇒ thread: [ name: T | k: '_;('_`(_`)(F, El)) | env: Env]

RULE thread: [k: •K] ⇒ •Bag

SYNTAX *Bool* ::= isDeclLabel (*KLabel*) [function]

RULE isDeclLabel (KL) ⇒ true when KL ==KLabel 'int_ orElseBool KL ==KLabel 'bool_ orElseBool KL ==KLabel 'void_ [anywhere]

SYNTAX *K* ::= isAlias (*Exp*) [function]

RULE isAlias (& X = E) ⇒ true [anywhere]

SYNTAX *Bool* ::= isLValue (*Exp*) [function]

RULE isLValue (lvalue (E)) ⇒ true [anywhere]

RULE isLValue (loc (L)) ⇒ true [anywhere]

RULE isLValue (X) ⇒ true [anywhere]

RULE isLValue (∗ X) ⇒ true [anywhere]

SYNTAX *Id* ::= getName (*Exp*) [function]

RULE getName (X) ⇒ X [anywhere]

RULE getName (& X) ⇒ X [anywhere]

RULE getName (∗ X) ⇒ X [anywhere]

RULE getName (KL(E)) ⇒ getName (E) when isDeclLabel (KL) ==Bool true [anywhere]

RULE getName (E1 = E2) ⇒ getName (E1) [anywhere]

RULE getName (KL('_`,_(E, •Exps))) ⇒ getName (E) when isDeclLabel (KL) ==Bool true [anywhere]

SYNTAX *K* ::= popb

RULE k: popb ⇒ •K | bstack: [Env] ⇒ •List | env: — ⇒ Env

END MODULE

MODULE CINK-SYNTAX

SYNTAX *Prop* ::= eqTo (*Id*, *Val*)

END MODULE

MODULE CINK

SYNTAX *#ModelCheckerState* ::= KItem (*Bag*)

SYNTAX *#Prop* ::= #eqTo (*Id*, *Val*) [function]

RULE eqTo (X, V) ⇒ #eqTo (X, V) [anywhere]

SYNTAX *Int* ::= val (*Bag*, *Id*) [function]

RULE val (T: [genv: X ↦ L | store: L ↦ I], X) ⇒ I [anywhere]

RULE KItem (B) LTL|= #eqTo (X, I) ⇒ true when val (B, X) ==K I [anywhere]

END MODULE