Project K Tools

January 16, 2019

Sârbu Iulia Iustina, grupa B1

Syntax

The K syntax of languages, calculi or systems, as well as the additional syntax needed for the semantics of these, is defined using context-free grammars (CFG) or, equivalently, algebraic signatures written using the mixfix notation (i.e., operation names include underscores as argument placeholders) [23 ,22,12]. We take the freedom to borrow from the algebraic universe any structures of interest on a by-need basis. In this paper we use *List*{*Nonterminal,terminal*} to refer to the nonterminal corresponding to *terminal*-separated lists of *Nonterminal*elements; for example,*List*{*Exp,@*}stands for @-separated lists of expressions. We skip the terminal when it is a comma; e.g.,*List*{*Exp*}stands for comma-separated lists of expressions.

module IMP-SYNTAX

Expressions

The syntax of expressions is identical to that in untyped SIMPLE, except for the logical conjunction and disjunction which have different strictness attributes, because they now have different evaluation strategies.

```
syntax AExp ::= Int ( builtin )
                | Id ( builtin )
                | String ( builtin )
                | ”read” ”(” ”)”
                | ”++” Id
                > left:
                AExp ”*” AExp [left]
                | AExp ”/” AExp [left]
                > left:
                AExp ”+” AExp [left, strict]
                | AExp ”-” AExp [left]
```

```
                    | "(" AExp ")" [bracket]
syntax BExp ::= Bool ( builtin )
                    | "!" BExp
                    | BExp "&&" BExp [non-assoc]
                    | AExp "<=" AExp [strict]
                    | AExp "<" AExp
                    | "(" BExp ")" [bracket]
```

Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameter. Unlike in untyped SIMPLE, all statement constructs which have arguments and are not desugared are strict, including the conditional and the while. Indeed, from a typing perspective,they are all strict: first type their arguments and then type the actual construct.

```
syntax Block ::= "{" Stmt "}"
                    | "{" "}"
syntax Stmt ::= Block
                    | "int" Id ";"
                    | "print" "(" AExp ")" ";" [strict]
                    | Id "=" AExp ";" [strict(2)]
                    | "if" BExp Block "else" Block [strict(1)]
                    | "while" BExp Block
                    > Stmt Stmt [right]
```

Static semantics

Here we define the type system of SIMPLE. Like concrete semantics, type systems defined in K are also executable. However,K type systems turn into type checkers instead of interpreters when executed. The typing process is done in two (overlapping) phases. In the first phase the global environment is built, which contains type bindings for all the globally declared variables and functions. For functions, the declared types will be *trusted* during the first phase and simply bound to their corresponding function names and placed in the global type environment. At the same time, type-checking tasks that the function bodies indeed respect their claimed types are generated. All these tasks are verified during the second phase. This way, all the global variable and function declarations are available in the global type environment and can be used to type-check each function body. This is consistent with the semantics of untyped SIMPLE, where functions can access all the global variables and can call any other function declared in the same program. The two phases may overlap because of the K concurrent semantics. For example, a function task can be started while the first phase is still running; moreover, it may even complete before the first phase does, namely when all the global variables and functions that it needs have already been processed and made available in the global environment by the first phase task.

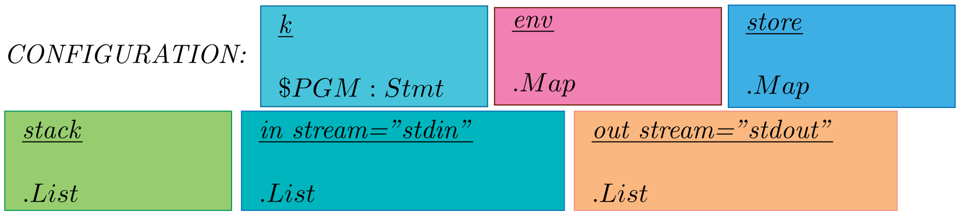
module IMP
imports IMP-SYNTAX

```
syntax KResult ::= Bool
                 | Int
                 | String
```

Configuration

The configuration of our type system consists of a tasks cell holding various typing task cells, and a global type environment.

CONFIGURATION:



Common expression constructs

The rules below are straightforward and self-explanatory:

RULE
I1 *int* <= I2 *int* => I1 <= *int* I2

RULE
I1 *int* + I2 *int* => I1 +*int* I2

RULE
S1 *stmt* S2 *stmt* => S1 ~> S2

RULE

k

intX: Id; => . . . .

env

M: Map=> M[X< - ! L: *Int*]

store

M": Map(. Map=> ! L|− >0)

RULE

k

(X: Id= V: *Int* ; => .
) . . .

env

. . . X|− >L: *Int* . . .

store

. . . L|− >(_=> V) . .
.

RULE

k

(X: Id=> V) . . .

env

. . . X|− >L: *Int* . . .

store

. . . L|− >V: *Int* . . .

RULE
IF true B1 *block* else B2 *block* => B1

RULE
IF false B1 *block* else B2 *block* => B2

syntax KItem ::= "restoreEnv"

RULE

k

{ S: *Stmt* } => S~> restoreEnv. . .

env

Env

stack

(. List=> ListItem(Env)) . . .

RULE

k

(restoreEnv=> .) . . .

env

_=> Env: Map

stack

(ListItem(Env) => . List) . . .

RULE
{ } => .

RULE
WHILE *BExp* { *stmt* } => IF { WHILE { } }
else { }

RULE

k

(+ + X: Id=> V+*int* 1) . . .

env

. . . X|− >L. . .

store

. . . (L: *Int* |− >(V: *Int* => V+*int* 1)) . . .

RULE

<u>*k*</u>

(read( ) => V) . . .

<u>*In*</u>

(ListItem(V: *Int*) => . List) . . .

Input/Output

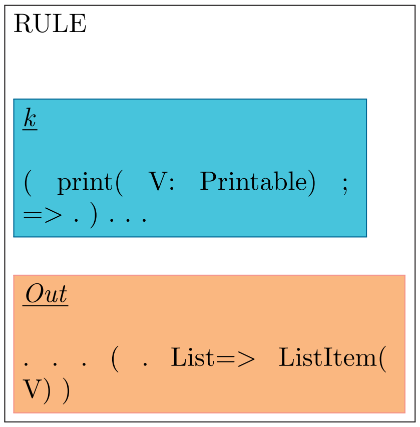
The read expression construct types to an integer, while print types to a statement provided that all its arguments type to integers or strings.

```
syntax Printable ::= Int
                   | String
```

Expressions

The syntax of expressions is identical to that in untyped SIMPLE, except for the logical conjunction and disjunction which have different strictness attributes, because they now have different evaluation strategies.

```
syntax AExp ::= Printable
```


RULE

k

(print(V: Printable) ; => .) . . .

Out

. . . (. List=> ListItem(V))

Conclusion

The K semantic framework, consisting of a general-purpose concurrent rewriting approach together with a definitional technique specialized for concurrent programming languages and systems, brings together the advantages of existing language definitional frameworks while avoiding their limitations. In spite of its youth, the K framework has already proven to be practical, as it has been used with relatively little effort to define complex languages like Java, Scheme, Verilog, or C, and to use those definitions for analyzing programs written in those languages. K is currently under heavy development, with bugs being fixed and new features and capabilities added on a regular basis.