SIMPLE — Typed — Dynamic

Grigore Roșu and Traian Florin Șerbănuță {grosu,tserban2}@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

This is the K dynamic semantics of the typed SIMPLE language. It is very similar to the semantics of the untyped SIMPLE, the difference being that we now dynamically check the typing policy described in the static semantics of typed SIMPLE. Consequently, we can now safely erase types after type checking and execute programs without types. Erasing types paves the route semantics, such as memory leak due to leaving an array out of bounds. We will highlight the differences between the dynamically typed and the untyped SIMPLE as we proceed with the semantics. We recommend the reader to consult the untyped SIMPLE semantics for a detailed discussion on the common parts of the typed SIMPLE language.

MODULE SIMPLE-TYPED-DYNAMIC-SYNTAX

Syntax

The syntax of typed SIMPLE extends that of untyped SIMPLE with support for declaring types in variables and functions.

The syntax below is identical to that of the static semantics of typed SIMPLE. However, the K statement attribute use this time is for dynamic typechecking, to update the dynamic evaluation strategies of the various language constructs.

```
syntax AExp ::= Value[ "over" ]
```

```
syntax Value ::= List{Name,","}
```

```
syntax Decl ::= Type Id
              | Type Id "[" AExp "]" Decl
```

Type

```
syntax Type ::= void
              | int
              | bool
              | string
              | Type "*" Type
              | Type "->" Type
              | Type "[]"
```

```
syntax Types ::= List{Type,","}
```

Declaration

```
syntax Params ::= Type Id
```

```
syntax Params ::= List{Params,","}
```

```
syntax Decl ::= Type Id ";"
              | Type Id "=" AExp ";" [strict(2)]
```

Expressions

```
syntax Exp ::= Id
             | Value
             | String
             | "(" Exp ")" [bracket]
             | Exp "(" Exps ")" [strict(2), result(K)]
             | Exp "[" Exp "]" [strict]
             | "sizeof" "(" Type ")"
             | "sizeof" "(" Exp ")"
             > Exp "++" [strict]
             | Exp "--" [strict]
             | "++" Exp [strict]
             | "--" Exp [strict]
             > Exp "*" Exp [strict]
             | Exp "/" Exp [strict]
             | Exp "%" Exp [strict]
             > Exp "+" Exp [strict]
             | Exp "-" Exp [strict]
             > Exp "<" Exp [strict]
             | Exp "<=" Exp [strict]
             | Exp ">" Exp [strict]
             | Exp ">=" Exp [strict]
             > Exp "==" Exp [strict]
             | Exp "!=" Exp [strict]
             > Exp "&&" Exp [strict(1)]
             > Exp "||" Exp [strict(1)]
             > right:
               Exp "=" Exp [strict(2)]
```

Like in the static semantics, there is no need for lists of identifiers (because we now have lists of parameters).

```
syntax Exps ::= List{Exp,","} [seq]
```

Statements

```
syntax Block ::= "{" "}"
               | "{" Stmt "}"
```

```
syntax Stmt ::= Decl
              | Block
              | "if" "(" Exp ")" Block [strict(1)]
              | "if" "(" Exp ")" Block "else" Block [strict(1)]
              | "while" "(" Exp ")" Block
              | "for" "(" Exp ";" Exp ";" Exp ")" Block
              | "return" Exp ";" [strict]
              | "return" ";"
              | "try" Block "catch" "(" Params ")" Block
              | "throw" Exp ";" [strict]
              | "assert" "(" Exp ")" ";" [strict]
              | "printf" "(" Exp ")" ";" [strict]
              | "scanf" "(" Exp ")" ";" [strict]
              | Exp ";" [strict]
```

```
syntax Stmt ::= Stmt
              | Stmt Stmt
```

The name-decaying macros like in the statically typed SIMPLE.

```
rule _:Id [ I:Int ] => *(_ + I)                                                    [macro]
```

```
rule _:Id [ I:Int , J:Int ] => *(_ + I * lookup(tail(types[_]), J))                [macro]
```

```
rule _:Id [ I:Int , J:Int , K:Int ] => *(_ + I * lookup(tail(types[_]), J) * K)    [macro]
```

```
rule _:Id [ _ , _ , _ , _ ]
  => *(_ + 1 * 2 * 3 * 4)                                                          [macro]
```

END MODULE

MODULE SIMPLE-TYPED-DYNAMIC

Semantics

Values and results

These are similar to those of untyped SIMPLE, except that the array references and the function abstractions now also hold information about the types of their elements/results (namely the type of an index in the heritage) into the dynamic type of operations at the end of the module.

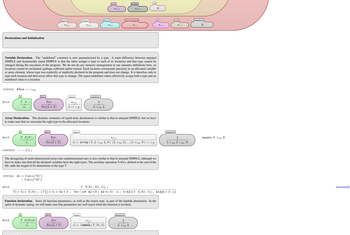
```
syntax Val ::= Int
             | Bool
             | String
             | Array Type Int Int
             | Lambda Params Stmt [binder]
```

```
syntax Vals ::= List{Val,","}
```

```
syntax KResult ::= Val
```

Configuration

The configuration is almost identical to that of untyped SIMPLE, except for a return cell inside the control cell. This return cell is used to hold the value returned by a function, so that it can be passed to the caller after all the pending local cells are cleaned. The existence of the cell will fail for a return if a function is invoked out of the context wherein the evaluation of the function's body is supposed to return a value.



Declaration and Initialization

Variable Declaration. The "variables" construct is now parameterized by a type. A main difference between untyped SIMPLE and typed SIMPLE is that in the latter, the type of a variable is stored in the environment, and it does not change during the execution of the program. We do not do, anymore, consequently to non-trivial modifications like the location counter for incremental, garbage collected and/or moved. Each location corresponds precisely to an allocated variable and it is never reused. Consequently, we can safely store the type of a variable in the environment, and we can safely erase it from the store and replace it with the value of the variable. This speed and memory efficient scope holds in type and in untyped SIMPLE.

```
syntax KItem ::= Type
```

Array Declaration. The dynamic semantics of typed array declarations is similar to that in untyped SIMPLE, but we have to take care that we associate the right type to the declared array.

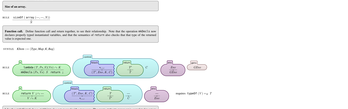
```
context int _:Id [ HOLE ] ;
```

The dereferencing of multi-dimensional arrays uses nondeterministic rules in a similar to that in untyped SIMPLE, although we have to make sure that all the declared variables have the right types. The auxiliary operation Prefix defined at the end of the module is used for this purpose.

```
syntax AExp ::= Value "[" Vals "]"
              | Value "[" Vals "]" "=" AExp [strict(2)]
```

```
rule V:Value [ Vs:Vals , I:Int ] => V [ Vs ] [ I ]
  requires Prefix(Vs, types[V]) =/=K .List
```

Function declaration. Store all function parameters, as well as the return type, as part of the lambda abstraction. To be safe in all dynamic typing, we will make sure that parameters are well typed when the function is called.



Calling main()

When done with the first pass, call main().

```
syntax KItem ::= execute
```

Expressions

Variable lookup

Variable/Array increment.

Arithmetic operators.

```
rule -I:Int => 0 -Int I
rule ++ V:Value => V + 1
rule V:Value ++ => V
rule -- V:Value => V - 1
rule -- V:Value => V - 1
rule I1:Int / I2:Int => I1 /Int I2
  requires I2 =/=Int 0
rule I1:Int % I2:Int => I1 %Int I2
  requires I2 =/=Int 0
rule I1:Int + I2:Int => I1 +Int I2
rule I1:Int - I2:Int => I1 -Int I2
rule I1:Int < I2:Int => I1 <Int I2
rule I1:Int <= I2:Int => I1 <=Int I2
rule I1:Int > I2:Int => I1 >Int I2
rule I1:Int >= I2:Int => I1 >=Int I2
rule I1:Int == I2:Int => I1 ==Int I2
rule I1:Int != I2:Int => I1 =/=Int I2
rule ! I:Int => notBool I
rule true && B:Bool => B
rule false && _ => false
rule true || _ => true
rule false || B:Bool => B
```

String Indexing. Check array bounds, as part of the dynamic typing policy.

```
rule S:String [ I:Int ]
  => substrString(S, I, I +Int 1)
  requires 0 <=Int I andBool I <Int lengthString(S)
rule lookup(string(_,_,S),I) => S[I]
  requires 0 <=Int I andBool I <Int lengthString(S)
```

Size of an array.

```
rule sizeOf(array(_,_,_)) => 0
```

Function call. Define functions call and return together, in one short relationship. Note that the operation shiftIn(I) now shares properly typed (nonempty) variables, and that the semantics of return also checks that type of the returned value is expected one.

Like the undefined behavior, misflowing also goes tagged with a type now. The empty return/no statement is completed in order for the following rules tagged to operate.

```
syntax Val ::= noThing(Type)
```

Exit

Assignment. The assignment now checks that the type of the assigned location is preserved.

```
context _:Id = HOLE ;
```

Statements

Blocks.

```
rule {}
  => .
rule { S:Stmt }
  => S
```

Sequential composition.

```
rule .
  => .
rule S1:Stmt S2:Stmt
  => S1
  ~> S2
```

Expression statements.

```
rule V:Value ;
  => .
```

Conditional.

```
rule if(true) S:Stmt => S
rule if(false) _ => .
```

While loop.

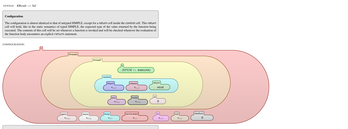
```
rule while(B) S
  => if(B) { S while(B) S } else {}
```

Print. We only allow printing integers and strings.

```
rule printf(I:Int) => printInt(I)
rule printf(S:String) => printString(S)
```

Exceptions. Exception parameters are now typed, but note that the semantics below works correctly only when the thrown exception has the same type as the statement in a catch parameter. To keep things simple, but the latter being we can assume that SIMPLE code does not have useless dead code, we think we can live with the temporary breach in the rule.

```
syntax KItem ::= throw(Int, Map, Map)
```



Threads.

Thread creation

Thread termination

Thread joining

Scalpels lock.

Kitchen lock.

Rendezvous synchronization

Auxiliary declarations and operations

Types. Three a list of parameters and a list of instance values for them into a list of variable declarations.

```
syntax Decl ::= shiftIn(Params,Vals) [function]
```

```
rule shiftIn(T X:Id, Ps:Params, V:Val, Vs:Vals)
  => T X:Id = V; shiftIn(Ps,Vs)
rule shiftIn(.Params,.Vals) => .
```

Location lookup.

```
syntax KItem ::= lookup(Int)
```

Environment recovery

```
syntax KItem ::= env(Map)
```


Undo and bye

```
syntax Exp ::= UnDef(K)
```

```
syntax Val ::= Int [tok]
```

```
context UnDef(HOLE)
context lookup(HOLE)
rule UnDef(V:Val) => V
```

Shifts the corresponding Stack to an array type

```
syntax Type ::= Type "Stack" [function]
```

```
rule T:Type Stack
  => T [] Stack
rule .Types Stack => .Types
```

Sequences of locations.

```
syntax Map ::= Int "..." Int [function]
```

```
rule I ... I => I |-> I
rule I ... J => I |-> I (I +Int 1) ... J
  requires I <Int J
```

```
syntax Set ::= keys(Map) [function]
```

```
rule keys(.Map) => .Set
rule keys(K |-> _ M) => SetItem(K) keys(M)
```

Lists of types in a parameter.

```
syntax Types ::= getTypes(Params) [function]
```

```
rule getTypes(.Params) => .Types
rule getTypes(T X:Id, Ps:Params) => T, getTypes(Ps)
rule getTypes(T X:Id [ I:Int ], Ps:Params) => T [ I ], getTypes(Ps)
rule getTypes(T X:Id [ I:Int , J:Int ], Ps:Params) => T [ I , J ], getTypes(Ps)
rule getTypes(T X:Id [ I:Int , J:Int , K:Int ], Ps:Params) => T [ I , J , K ], getTypes(Ps)
```

```
rule sizeOf(_:Id [ I:Int ]) => I
rule sizeOf(_:Id [ I:Int , J:Int ]) => I * J
rule sizeOf(_:Id [ I:Int , J:Int , K:Int ]) => I * J * K
```

Prefix of lists.

```
syntax Bool ::= Prefix(Types,Types) [function]
```

```
rule Prefix(.Types,_) => true
rule Prefix(_,.Types) => false
rule Prefix(T1,Ts1) => T1 ==K T2 andBool Prefix(Ts1,Ts2)
```

END MODULE