SIMPLE — Typed — Dynamic

Grigore Roşu and Traian Florin Şerbănuţă (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. Variables are now tuples consisting of their value and their type, and the assignment operation checks that the value and the type stored in a variable match and in assignments blocks there is a mixing of static and dynamic. 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 Syntax page with the syntax of typed SIMPLE and the semantics of the untyped SIMPLE language.

MODULE SIMPLE-TYPED-DYNAMIC-SYNTAX

Syntax

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

The syntax below is identical to that of the static semantics of typed SIMPLE. However, the K environment attributes are like those of the untyped SIMPLE, to support the desired evaluation strategies of the various language constructs.

```
syntax K ::= Decl [Decl]
```

Types

```
syntax Type ::= void
                int
                bool
                char
                float
                Id[Type]
                Type*[Type]
                Type[Int*]

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

Declarations

```
syntax Params ::= Type Id*

syntax Decls ::= List{Decls, ";"}

syntax Decl ::= Type Id; |
                Type Id(Params){K}
```

Expressions

```
syntax Exp ::= Id
               Int
               Bool
               Char
               Float[Int]
               Id(Exps)
               Exp[Exp]
               *Exp[Int]
               Id[Int*][Exp*][Int*]
               cast(Type, Exp)
               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)]
               Exp ? Exp : Exp[strict(1)]
               Exp, Exp[strict(1)]
               Id = Exp[strict(2)]
               Id[Int*] = Exp[strict(2)]
               Id[Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
               Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp[strict(2)]
```

Like in the untyped SIMPLE, there is bit and bit-line of identifiers (because we now have lists of parameters).

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

Statements

```
syntax Block ::= {}
                 {Stmt*}

syntax Stmt ::= Block
                ;
                if (Exp) Block else Block [strict(1), avoid]
                while (Exp) Block
                while (Exp) Block
                for (Stmt; Exp; Exp) Block
                break; | continue;
                return Exp; [strict]
                return;
                try Block catch (Decl) Block
                throw Exp; [strict]
                switch Exp {Cases}
                Type Id[Int*]; [strict(2)]
                return Exp; [strict]
                Exp; [strict]
                Id = Exp; [strict(2)]
                Id[Int*] = Exp; [strict(2)]
                Id[Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
                Id[Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*][Int*] = Exp; [strict(2)]
```

The same designing choices like in the statically typed SIMPLE.

```
syntax Cases ::= Case | Cases Cases [left]

syntax Case ::= Int : Stmt* [strict(1)]
                default : Stmt*

syntax Decl ::= Decl* Stmt* [strict(1)]
                Decl* while(Exp)Stmt* Stmt* [strict(1)]

syntax Decl ::= Decl* Stmt* [strict(1)]
                Decl* while(Exp)Stmt* Stmt* [strict(1)]

syntax K ::= for (Decl*; Exp; Exp) Block [strict(1)]
             for (Decl*; Exp; Exp) Block [strict(1)]

syntax K ::= for (Decl*; Exp; Exp) Stmt* [strict(1)]
             for (Decl*; Exp; Exp) Stmt* [strict(1)]
```

END MODULE

MODULE SIMPLE-TYPED-DYNAMIC

Semantics

Values and vars

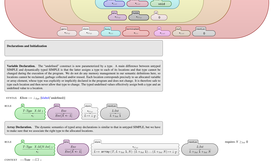
We define values in terms of untyped SIMPLE, except that the array references and the function abstractions are also kept like values. Other types are needed in order to easily compute the type of any value in the language (see the auxiliary typeof operation at the end of the module).

```
syntax Val ::= Int
               Bool
               Char
               array(Type, Int, Int)[Int*] [strict]
               fun(Type, Decls, Decls)[Int*] [strict]

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

syntax Exp ::= Val

syntax KResult ::= Val
```

Configuration

The configuration is almost identical to that of untyped SIMPLE, except for a return cell inside the control cell. The initial cell will hold, like in the latter semantics of untyped SIMPLE, the expected type of the value returned by the function being evaluated. The return cell will hold the value returned by the function, or it will wait until it hits the closed brace of the function to return the default return expression.

CONFIGURATION:


K
k
Int
nextloc
Int
K
return
K
K
K