SIMPLE — Typed — Static

Grigore Roşu and Traian Florin Şerbănuţă (grosu,tsarbanu2@illinois.edu)
University of Illinois at Urbana-Champaign

Abstract

This is the 2 definition of the static semantics of the typed SIMPLE language, or in other words, a type system for the typed SIMPLE language. The syntax of the language is presented in a way that allows for a direct translation into a parsing and static analysis tool of the language rules and definitions. We first write down the syntax and the static semantics of the language, and then we discuss some of the design decisions and some of the properties of the resulting type system.

When designing a type system for a language, no matter its what purpose, we have to decide upon the intended typing policy. Some of the properties that we want to have in a type system are: (1) the type system should be sound, (2) the type of a valid program which don't have a runtime error shouldn't be changed by other functions that do not cause a runtime error, (3) the model or other functions when type expects them to return a value (so, an error is not a value "recovery" mechanism, which returns a value of the type that was expected in the place where the error happened), (4) the type system should be simple, and (5) the type system should be easy to extend with new types (or at a minimum, it should be easy to extend a type system that do not have any of the properties listed above to a type system that has all the properties listed above).

On the other hand, you want your type system to have a reasonable performance when implemented, that will bring both the time and the space complexity of the type checker/inferer to a minimum. Unfortunately, the type system that satisfies all the previous desiderata is not so expensive to be feasible in practice. A simple polynomic when dropping parting policy is a good candidate for a type system that satisfies all the previous desiderata, and it is also feasible in practice.

Below we give the K type system of SIMPLE formally, we discuss, informally, the intended typing policy:

- Each primitive value has its own type, which can be (int, bool, or string). There is also a type for void for statement values, for example for the result of a function assign to certain or value that only be used for its side effects, like a function.
- The syntax of untyped SIMPLE is extended to allow type declarations for all the variables, including array variables. This is done in C-like syntax. For example, `int a = 5;` or `int arr[10];` or `int arr[] = {0,1,2,3,4,5,6,7,8,9};` (the syntax for array initialization is the same as in C, and it allows for the size of the array to be left out, in which case the dimensional array can contain arbitrary sequences, although here have the array of array semantics).
- Function declarations are also extended to allow for the type of the function parameters and the return value to be stated. For other functions, we also need function types. We will use the parenthesis notation to write down the function types, but will separate the argument types with commas. For example, a function returning an array of bool elements and taking two integers and a string as its parameters would be represented by the type `(int,int,string)[]bool`. Function types are also allowed in the syntax.

Like in untyped SIMPLE, some constructs can be designated into a another set of basic constructs. In general, it should be clear what a language does not type by looking at the top of the models with its non-sub-constructs.

MODULE SIMPLE-TYPED-SYNTAX' EXTENDS K

Syntax

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

```
SYNTAX Id ::= NAME
```

Types

Primitive, array and function types, as well as lists (or tuples) of such types. The lists of types are useful for function arguments.

```
SYNTAX Type ::= void
                int
                bool
                string
                Type[]
                (TypeList)
                Type [TypeList]

SYNTAX TypeList ::= List{Type,","}
```

Declarations

Variables and function declarations have the expected syntax. For variables, we basically just extended the rule for declaration of untyped SIMPLE with a type. For functions, besides replacing the Type1List keyword with a type, we also introduce a new syntactic category for typed variables. Events, and then we use it.

```
SYNTAX Decls ::= Type Id

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

SYNTAX Decl ::= Type Id;
                Type Id(DecsVarsOpt)
```

Expressions

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

```
SYNTAX Exp ::= Id
               Bool
               Int
               String
               Void
               <= Exp
               (Exp,Exp,List{Exp})
               Exp[Exp,List{Exp}]
               [Exp,List{Exp}]
               (Exp)
               Exp * Exp [seqstrict]
               Exp / Exp [seqstrict]
               Exp + Exp [seqstrict]
               Exp - Exp [seqstrict]
               Exp % Exp [seqstrict]
               Exp < Exp [seqstrict]
               Exp > Exp [seqstrict]
               Exp <= Exp [seqstrict]
               Exp >= Exp [seqstrict]
               Exp == Exp [seqstrict]
               Exp != Exp [seqstrict]
               Exp && Exp [seqstrict]
               Exp || Exp [seqstrict]
               Exp = Exp [seqstrict]
               Exp[Exp,List{Exp}] [seqstrict]
               [Exp,List{Exp}] [seqstrict]
```

Note that statement has not been declared void. This may seem unexpected, because the child thread shares the same resources, stack with the parent thread, so from a typing perspective the spawned statement under the same rules as a child thread of a function call. However, the spawned statement can be used in a place where a void value is expected, for example, in a child thread with the void statement, because those programs overall get stuck in the dynamic semantics. The type semantics of spawn forbids all such cases of programs.

We will need lists of expressions, defined below, but note that we do not need lists of identifiers anymore. They have been replaced by the list of parameters.

```
SYNTAX Exps ::= List{Exp,","} [seqstrict]
```

Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameters. Time and sleep statements in untyped SIMPLE all statements accepted a number of milliseconds, but in typed SIMPLE, we are adding the conditional and the wait for, Indeed, from a typing perspective, they act all strict. Are type their arguments and then type the actual statement.

```
SYNTAX Block ::= {}
                 {Stmts}

SYNTAX Stmt ::= Decl
                Block
                if(Exp)Block else Block [seqstrict, strict]
                while(Exp)Block
                while (Exp)Block [strict]
                for(Exp;Exp;Exp)Block
                return Exp; [strict]
                return;
                assign Id to Exp [strict]
                try Block catch(Decl)Block [strict(2)]
                print(Exp) [strict]
                print Exp; [strict]
                sleep(Exp); [strict]
                sleep Exp; [strict]
                wait for Exp [strict]
                Exp; [strict]
                ;
```

Note that the sequenced composition is now seqstrictly strict, because, unlike in the dynamic semantics where statements dissolved, they now reduce to the void type, which is a value.

```
SYNTAX Stmt ::= Stmt
                Stmt Stmts [seqstrict]
```

Designating sources

We use the same designating sources like in untyped SIMPLE, but, of course, including the types of the involved variables.

```
SYNTAX      ::= of Id
                if Id else Id

SYNTAX Var-Short List{Var-Short}[Bool]
                [Var-Short List{Var-Short}]
                [Var-Short List{Var-Short}, Bool]
                [ * Id | * Id, Id ]
                [ * Id, * Id, Id, Bool ]
SYNTAX Var Var-Short | Var-Short, Var
                [ * Id, * Id, Id, Bool, Var ]
                [ * Id, * Id, Id, Bool, Var, Var ]
SYNTAX Var ::= Var-Short
               Var-Short, Var-Short
               Var-Short, Var-Short, Var-Short
SYNTAX Var ::= Var-Short
               Var-Short, Var-Short
               Var-Short, Var-Short, Var-Short
```

MODULE SIMPLE-TYPED-SYNTAX'

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 type system of untyped SIMPLE was a simple one, with only one sort for the global environment is bool, which contains type bindings for all the globals, declared variables and functions. For functions, the declared types will be the type of the function, which will be a function type, for example, `int(int,int,int)[]` for a function that takes three integers and returns an array of integers. Variables and functions can shadowing under the function bodies, local variables and formal parameters are generated. All these rules are still valid in the typed version of SIMPLE, but we need to add some more rules. For example, we need to add a sort for the global type environment and a sort for types to type check function calls. This is convenient with the syntax of untyped SIMPLE, where functions are a sort of the global variables and can all only either function declared in the same scope, or built-in functions. In typed SIMPLE, we need to add a sort for the function types, and we need to add a sort for the function bodies, because we need to check whether the return type, returns values of the proper type. Because we want to be able to type check function calls, we need to add a sort for the function parameters, because we need to be able to type check the parameters of a function call.

```
SYNTAX Exp ::= Int
               Bool
               String
               Void
               <= Exp
               (Exp,Exp,List{Exp})
               Exp[Exp,List{Exp}]
               [Exp,List{Exp}]
               (Exp)
               Exp * Exp [seqstrict]
               Exp / Exp [seqstrict]
               Exp + Exp [seqstrict]
               Exp - Exp [seqstrict]
               Exp % Exp [seqstrict]
               Exp < Exp [seqstrict]
               Exp > Exp [seqstrict]
               Exp <= Exp [seqstrict]
               Exp >= Exp [seqstrict]
               Exp == Exp [seqstrict]
               Exp != Exp [seqstrict]
               Exp && Exp [seqstrict]
               Exp || Exp [seqstrict]
               Exp = Exp [seqstrict]
               Exp[Exp,List{Exp}] [seqstrict]
               [Exp,List{Exp}] [seqstrict]
```

Note that statement has not been declared void. This may seem unexpected, because the child thread shares the same resources, stack with the parent thread, so from a typing perspective the spawned statement under the same rules as a child thread of a function call. However, the spawned statement can be used in a place where a void value is expected, for example, in a child thread with the void statement, because those programs overall get stuck in the dynamic semantics. The type semantics of spawn forbids all such cases of programs.

We will need lists of expressions, defined below, but note that we do not need lists of identifiers anymore. They have been replaced by the list of parameters.

```
SYNTAX Exps ::= List{Exp,","} [seqstrict]
```

Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameters. Time and sleep statements in untyped SIMPLE all statements accepted a number of milliseconds, but in typed SIMPLE, we are adding the conditional and the wait for, Indeed, from a typing perspective, they act all strict. Are type their arguments and then type the actual statement.

```
SYNTAX Block ::= {}
                 {Stmts}

SYNTAX Stmt ::= Decl
                Block
                if(Exp)Block else Block [seqstrict, strict]
                while(Exp)Block
                while (Exp)Block [strict]
                for(Exp;Exp;Exp)Block
                return Exp; [strict]
                return;
                assign Id to Exp [strict]
                try Block catch(Decl)Block [strict(2)]
                print(Exp) [strict]
                print Exp; [strict]
                sleep(Exp); [strict]
                sleep Exp; [strict]
                wait for Exp [strict]
                Exp; [strict]
                ;
```

Note that the sequenced composition is now seqstrictly strict, because, unlike in the dynamic semantics where statements dissolved, they now reduce to the void type, which is a value.

```
SYNTAX Stmt ::= Stmt
                Stmt Stmts [seqstrict]
```

Designating sources

We use the same designating sources like in untyped SIMPLE, but, of course, including the types of the involved variables.

```
SYNTAX      ::= of Id
                if Id else Id

SYNTAX Var-Short List{Var-Short}[Bool]
                [Var-Short List{Var-Short}]
                [Var-Short List{Var-Short}, Bool]
                [ * Id | * Id, Id ]
                [ * Id, * Id, Id, Bool ]
SYNTAX Var Var-Short | Var-Short, Var
                [ * Id, * Id, Id, Bool, Var ]
                [ * Id, * Id, Id, Bool, Var, Var ]
SYNTAX Var ::= Var-Short
               Var-Short, Var-Short
               Var-Short, Var-Short, Var-Short
SYNTAX Var ::= Var-Short
               Var-Short, Var-Short
               Var-Short, Var-Short, Var-Short
```

END MODULE

MODULE SIMPLE-TYPED-SYNTAX'

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 type system of untyped SIMPLE was a simple one, with only one sort for the global environment is bool, which contains type bindings for all the globals, declared variables and functions. For functions, the declared types will be the type of the function, which will be a function type, for example, `int(int,int,int)[]` for a function that takes three integers and returns an array of integers. Variables and functions can shadowing under the function bodies, local variables and formal parameters are generated. All these rules are still valid in the typed version of SIMPLE, but we need to add some more rules. For example, we need to add a sort for the global type environment and a sort for types to type check function calls. This is convenient with the syntax of untyped SIMPLE, where functions are a sort of the global variables and can all only either function declared in the same scope, or built-in functions. In typed SIMPLE, we need to add a sort for the function types, and we need to add a sort for the function bodies, because we need to check whether the return type, returns values of the proper type. Because we want to be able to type check function calls, we need to add a sort for the function parameters, because we need to be able to type check the parameters of a function call.

```
SYNTAX Exp ::= Int
               Bool
               String
               Void
               <= Exp
               (Exp,Exp,List{Exp})
               Exp[Exp,List{Exp}]
               [Exp,List{Exp}]
               (Exp)
               Exp * Exp [seqstrict]
               Exp / Exp [seqstrict]
               Exp + Exp [seqstrict]
               Exp - Exp [seqstrict]
               Exp % Exp [seqstrict]
               Exp < Exp [seqstrict]
               Exp > Exp [seqstrict]
               Exp <= Exp [seqstrict]
               Exp >= Exp [seqstrict]
               Exp == Exp [seqstrict]
               Exp != Exp [seqstrict]
               Exp && Exp [seqstrict]
               Exp || Exp [seqstrict]
               Exp = Exp [seqstrict]
               Exp[Exp,List{Exp}] [seqstrict]
               [Exp,List{Exp}] [seqstrict]
```

Note that statement has not been declared void. This may seem unexpected, because the child thread shares the same resources, stack with the parent thread, so from a typing perspective the spawned statement under the same rules as a child thread of a function call. However, the spawned statement can be used in a place where a void value is expected, for example, in a child thread with the void statement, because those programs overall get stuck in the dynamic semantics. The type semantics of spawn forbids all such cases of programs.

We will need lists of expressions, defined below, but note that we do not need lists of identifiers anymore. They have been replaced by the list of parameters.

```
SYNTAX Exps ::= List{Exp,","} [seqstrict]
```

Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameters. Time and sleep statements in untyped SIMPLE all statements accepted a number of milliseconds, but in typed SIMPLE, we are adding the conditional and the wait for, Indeed, from a typing perspective, they act all strict. Are type their arguments and then type the actual statement.

```
SYNTAX Block ::= {}
                 {Stmts}

SYNTAX Stmt ::= Decl
                Block
                if(Exp)Block else Block [seqstrict, strict]
                while(Exp)Block
                while (Exp)Block [strict]
                for(Exp;Exp;Exp)Block
                return Exp; [strict]
                return;
                assign Id to Exp [strict]
                try Block catch(Decl)Block [strict(2)]
                print(Exp) [strict]
                print Exp; [strict]
                sleep(Exp); [strict]
                sleep Exp; [strict]
                wait for Exp [strict]
                Exp; [strict]
                ;
```

Note that the sequenced composition is now seqstrictly strict, because, unlike in the dynamic semantics where statements dissolved, they now reduce to the void type, which is a value.

```
SYNTAX Stmt ::= Stmt
                Stmt Stmts [seqstrict]
```

Designating sources

We use the same designating sources like in untyped SIMPLE, but, of course, including the types of the involved variables.

```
SYNTAX      ::= of Id
                if Id else Id

SYNTAX Var-Short List{Var-Short}[Bool]
                [Var-Short List{Var-Short}]
                [Var-Short List{Var-Short}, Bool]
                [ * Id | * Id, Id ]
                [ * Id, * Id, Id, Bool ]
SYNTAX Var Var-Short | Var-Short, Var
                [ * Id, * Id, Id, Bool, Var ]
                [ * Id, * Id, Id, Bool, Var, Var ]
SYNTAX Var ::= Var-Short
               Var-Short, Var-Short
               Var-Short, Var-Short, Var-Short
SYNTAX Var ::= Var-Short
               Var-Short, Var-Short
               Var-Short, Var-Short, Var-Short
```

END MODULE

Configuration

The configuration of our type system consists of a tuple(s) of building cells(or sections) typing both cells, and a global type environment. Each tuple includes a cell defining the node in ast, a form cell holding the local type environment, and a cell holding the type of the expression. Indeed, the global type environment is a map, a form cell holding the local type environment, and a cell holding the type of the expression. Indeed, the global type environment is a map, a form cell holding the local type environment, and a cell holding the type of the expected type