SIMPLE — Typed — Static

Grigore Roşu and Traian Florin Şerbănuţă ({grosu,tserban2}@illinois.edu)
University of Illinois at Urbana-Champaign

Abstract

This is the $\mathbb{K}$ definition of the static semantics of the typed SIMPLE language, or in other words, a type system for the typed SIMPLE language in $\mathbb{K}$. We do not re-discuss the various features of the SIMPLE language here. The reader is referred to the untyped version of the language for such discussions. We here only focus on the new and interesting problems raised by the addition of type declarations, and what it takes to devise a type checker for the language.

When designing a type system for a language, no matter within what paradigm, we have to decide upon the typing policy that we intend to capture by our type system. Note that we can have multiple type systems for the same language, one for which typing policy. For example, should we accept programs which don't have a main function? Or should we allow functions that do not return explicitly? Or should we allow functions whose type expects them to return a value (say an `int`) to use a plain "`return;`" statement, which returns no value, like in C? And so on and so forth. Typically, there are two opposite tensions when designing a type system. On the one hand, you want your type system to be as permissive as possible, that is, to accept as many programs that do not get stuck when executed with the untyped semantics as possible; this will keep the programmers using your language happy. On the other hand, you want your type system to have a reasonable performance when implemented; this will keep both the programmers and the implementers of your language happy. For example, a type system that rejects programs that could perform division-by-zero is not feasible in general. A simple guideline when designing typing policies is to imagine how the semantics of the untyped language may get stuck and try to prevent those situations from happening.

Before we give the $\mathbb{K}$ type system of SIMPLE formally, we discuss, informally, the intended typing policy:

- Each program should contain a `main` function. Indeed, the untyped SIMPLE semantics will get stuck on any program which does not have a `main` function.
- Each primitive value has its own type, which can be `int` `bool`, or `string`. There is also a type `void` for nonexistent values, for example for the result of a function meant to return no value (but only be used for its side effects, like a procedure).
- The syntax of untyped SIMPLE is extended to allow type declarations for all the variables. This is done in a Pascal-style, following the declared variable with a colon followed by the type. For example, "`var x:int;`" or "`var x:int=7, y:int, z:int=x+y;`".
- Arrays of values of type T have the type `array of` T and are declared using a syntax similar to the one for simple variables, but where the colon and the type follow the dimension of the declared array. For example, "`var x[10] : array of int;`", or "`var x[10,20] : array of array of int;`", or even "`var x : array of array of int;`" when `x` is only needed as a reference to an array (allocated somewhere else), or even "`var x[10] : array of array of int;`", as well as any combinations of simple variables (with or without initializations) and arrays (with or without allocation—given sizes).
- Functions taking arguments of type Ts (a list of types) and returning a result of type T have the type `function from` Ts `to` T (displayed as $Ts \to T$ in this generated PDF documentation). For example, a function taking an array of functions from `int` to `int` and returning an array of `bool` elements is declared using a syntax of the form

```
function f(x : array of function from int to int) : array of bool {
  ...
}
```

  and has the type `function from array of function from int to int to bool`.
- We allow any variable declarations at the top level. Functions can only be declared at the top level. No other statements are allowed at the top level. In particular we don'l allow declared variables to be initialized. That is because our semantics of initialized variables is to first declare them and then initialize them using an assignment statement; however, assignments are not allowed at the top level in typed SIMPLE. If you want to allow initialization for declared variables at the top level, then you have to do it explicitly in the semantics. For simplicity we don't. Each function can only access the other functions and variables declared at the top level, or its own locally declared variables. SIMPLE has static scoping.
- The various expression and statement constructs take only elements of the expected types.
- Increment and assignment can operate both on variables and on array elements. For example, if `f` has type `function from int to array of array of int` and function `g` has the type `function from int to int`, then the increment expression `++f(7)[g(2),g(3)]` is valid.
- The `for` loops only iterate over counter variables of type `int`, which therefore need not be manually declared; they are automatically assumed declared only for the scope of the `for` and of type `int`.
- Functions should only return values of their declared result type. To allow more flexibility to the programmers, we allow functions to use "`return;`" statements to terminate without returning an actual value, or to not explicitly use any return statement, regardless of their declared return type. This flexibility can be handy when writing programs using certain functions only for their side effects.
- For simplicity, we here limit exceptions to only throw integer values. This way, we don't need to declare a type for the variable that binds the thrown value (similarly to the counter variables in `for` loops).

Like in untyped SIMPLE, some constructs can be desugared into a smaller set of basic constructs.

MODULE SIMPLE-TYPED-STATIC-SYNTAX

Syntax

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

SYNTAX *#Id* ::= `main`

Types

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

```
SYNTAX  Type ::= int
               | bool
               | string
               | void
               | array of Type
               | Types → Type
SYNTAX  Exps ::= List{Exp,","}
SYNTAX  Types ::= List{Type,","}
```

Declarations

Variable and function declarations are allowed to have a more generous syntax than how we want them to be used in programs, but the type system will be defined in such a way that all abuses will be caught. For example, functions will only be allowed to take typed identifiers as parameters. The reason we prefer to allow a more generous syntax is to simplify our overall syntax by defining fewer syntactic categories. Recall that, after all, the syntax one defined in $\mathbb{K}$ definition is what we call "the syntax of the semantics", that is, some syntax which is convenient enough for users to write their desired semantic rules. This syntax is not meant to be used to parse complex programming language, such as C or Java. While $\mathbb{K}$'s syntax is good enough to parse simple and pedagogical languages like the ones discussed in this class, in practice one is expected to use external parsers for complex languages.

```
SYNTAX  Decl ::= var Exps ;
               | function #Id ( Exps ) : Type  Stmt
```

Expressions

The syntax of expressions is identical to that in untyped SIMPLE, except for the last construct in the sequence below. That is allowed exclusively only for parsing declarations as described above. It will be given no semantics.

```
SYNTAX  Exp ::= #Int
              | #Bool
              | #String
              | #Id
              | ++ Exp
              | Exp + Exp [strict]
              | Exp - Exp [strict]
              | Exp * Exp [strict]
              | Exp / Exp [strict]
              | Exp % Exp [strict]
              | - Exp [strict]
              | Exp < Exp [strict]
              | Exp <= Exp [strict]
              | Exp > Exp [strict]
              | Exp >= Exp [strict]
              | Exp == Exp [strict]
              | Exp != Exp [strict]
              | Exp and Exp [strict]
              | Exp or Exp [strict]
              | not Exp [strict]
              | Exp [ Exps ] [strict]
              | sizeOf( Exp ) [strict]
              | Exp ( Exps ) [strict]
              | read()
              | Exp = Exp [strict(2)]
              | Exp : Type
```

Statements

The statements have the same syntax as in untyped SIMPLE. That is because we decided that counters in `for` loops and values as exceptions can only be integers, so there is no need to declare them so (we will assume that in the semantics of these language constructs). Note that, 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  Stmt ::= {}
               | { Stmts }
               | Exp ; [strict]
               | if Exp then Stmt else Stmt [strict]
               | if Exp then Stmt
               | while Exp do Stmt [strict]
               | for #Id = Exp to Exp do Stmt
               | return Exp ; [strict]
               | return;
               | print( Exps ); [strict]
               | try Stmt catch( #Id ) Stmt [strict(1)]
               | throw Exp ; [strict]
               | spawn Stmt [strict]
               | acquire Exp ; [strict]
               | release Exp ; [strict]
               | rendezvous Exp ; [strict]
SYNTAX  Stmts ::= Decl
                | Stmt
                | Stmts  Stmts [seqstrict]
```

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

MACRO `if` E `then` S = `if` E `then` S `else {}`

MACRO `for` X = E_1 `to` E_2 `do` S = { `var` X : `int` = E_1 ; `while` X <= E_2 `do` { S X = X + 1 ; } }

MACRO `var` E_1 , E_2 , Es ; = `var` E_1 ; `var` E_2 , Es ;

MACRO `var` X : T = E ; = `var` X : T ; X = E ;

END MODULE

MODULE SIMPLE-TYPED-STATIC
IMPORTS SIMPLE-TYPED-STATIC-SYNTAX

Type System

Here we give the type system of SIMPLE using $\mathbb{K}$. Like concrete semantics, type systems defined in $\mathbb{K}$ are also executable. However, $\mathbb{K}$ 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 (concurrently) 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 in order to type-check each function code. This is consistent with the semantics of untyped SIMPLE, where functions can access all the global variables and can call any other functions declared in the same program. The two phases may overlap because of the $\mathbb{K}$ 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.

Extended syntax and results

The idea is to start with a configuration holding the program to type in one of its cells, then apply rewrite rules on it mixing types and language syntax, and eventually obtain a type instead of the original program. In other words, the program "evaluates" to its type using the $\mathbb{K}$ rules giving the type system of the language. In doing so, additional typing tasks for function bodies are generated and solved the same way. If this rewriting process gets stuck, then we say that the program is not well-typed. Otherwise the program is well-typed (by definition).

We start by allowing types to be used inside expressions and statements in our language. This way, types can be used together with language syntax in subsequent $\mathbb{K}$ rules without any parsing errors. Also, since programs and fragments of program will "evaluate" to their types, in order for the strictness and context declarations to be executable we state that types are results.

SYNTAX *Exp* ::= *Type*
SYNTAX *Stmt* ::= *Type*
SYNTAX *KResult* ::= *Type*

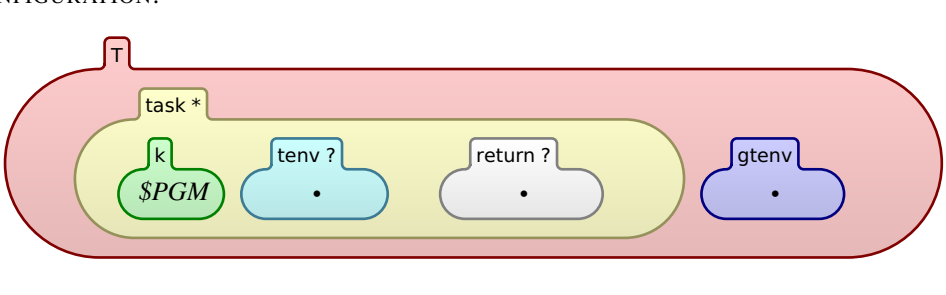
Configuration

The configuration of our type system consists of `task` cells and a global type environment. Each task includes a `k` cell holding the code to type and two optional cells:

- A `tenv` cell holding the local type environment.
- A `return` cell holding the return type of the currently checked function. This is needed in order to check whether return statements return values of the expected type.

The original program is put in a task containing no return or type environment cells (not that the multiplicity of these cells is "?", which means that they are not automatically included in the initial configuration.

CONFIGURATION:



Variable declarations

Variable declarations type as statements, that is, they "evaluate" to the type `stmt`. We did not need the `stmt` type as part of the typed SIMPLE syntax (indeed, users are not allowed to use the statement type explicitly), so we define it now. There are three cases that need to be considered: when the list of variables is empty (which can appear when functions have no arguments, since we reduce the typing of functions to typing variable declarations and statements—see below), when a simple variable is declared, and when an array variable is declared. The macros at the end of the syntax module above take care of reducing other variable declarations, including ones where the declared variables are initialized, to only these three cases. The first case is trivial and the second and third make use of a `bindto` helper operation, which takes a variable and a type and performs the actual binding in the current type environment cell when it reaches the top of the computation. Note that `bindto` applies the binding to the local type environment when that exists; otherwise it applies it to the global type environment. The third case requires an additional check, namely that the depth of the declared dimension type is smaller than or equal to the depth of the declared type (it can be strictly smaller, e.g., when we want the declared array to hold array references). The auxiliary operations (e.g., `checkDepth` and `bindto`) are defined at the end of the module, as usual.

SYNTAX *Type* ::= `stmt`
RULE `var;` ⇒ `stmt` [structural]

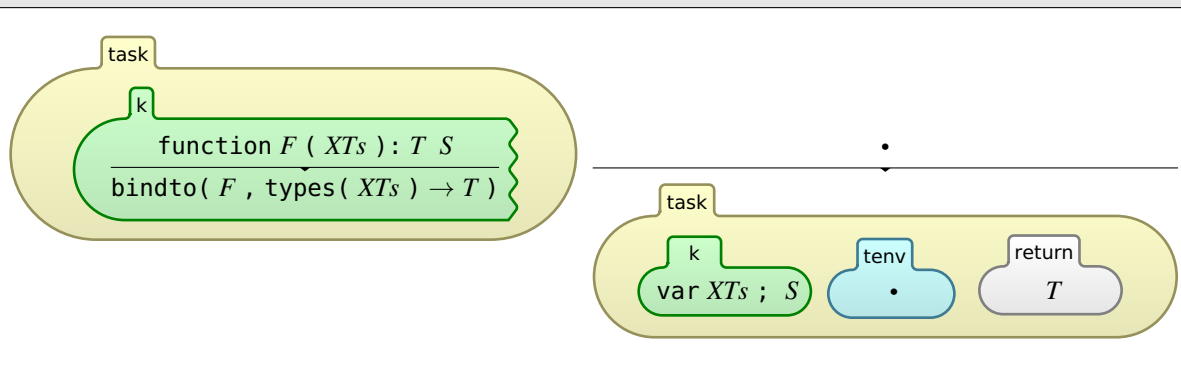
RULE `var` X : T ; ⇒ `bindto(` X , T `)`

CONTEXT: `var` – [□] : – ;

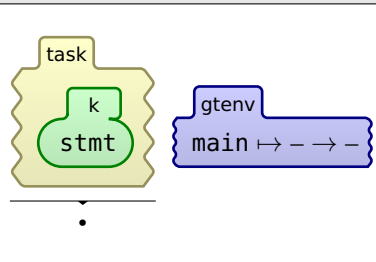
RULE `var` X [Ts] : T ; ⇒ `checkDepth(` Ts , T `)` ↷ `bindto(` X , T `)`

Function declarations

Functions are allowed to be declared only at the top level, indicated in the rule below by the fact that the `task` cell holds only a `k` cell. Indeed, as the rule below shows, generated function body tasks also contain `tenv` and `return` cells. Each function declaration adds a binding of its name to its declared function type in the current (in this case the global) type environment, but also adds a task into the `tasks` cell. The task consists of a typing a the statement declaring all the function parameters followed by the function body, together with the expected return type of the function. The code of the task makes use of other language constructs (variable declaration and sequential composition), so it is not very modular, but it is more compact and easier to understand than a more direct semantics. The auxiliary `types` operation, defined at the end of this module, will ensure that all the "expressions" in XTs are actually nothing but typed identifiers.

RULE


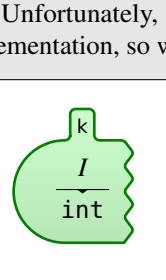
Once a task is completed, indicated by the fact that its `k` cell holds only the type `stmt`, we can dissolve its corresponding cell. Since the task may be the original one and since we want to enforce that programs include a main function, we also perform a check for `main` in the global type environment. This way, there should be no task cell left in the configuration when the program correctly type checks.

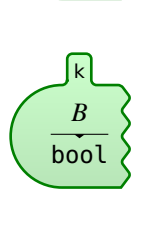
RULE


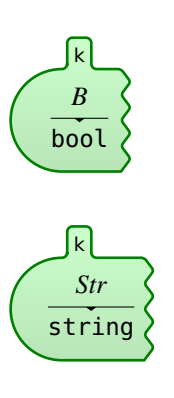
Expressions

Now that the entire machinery of the type system is operational, it is straightforward to type the various language constructs.

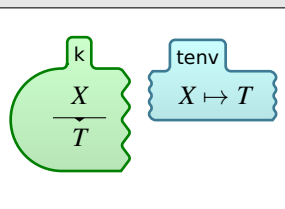
In theory, the first three rewrite rules below can apply anywhere to rewrite values into their types, not only at the top of the `k` cell. Unfortunately, since the $\mathbb{K}$ tool is implemented also by rewriting, that would get into conflict with the internals of our implementation, so we restrict their application to the top of the `k` cell.

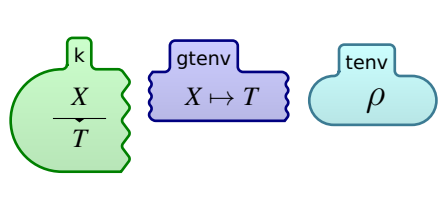
RULE


RULE


RULE


The are two cases to distinguish for variable lookup: if the variable is bound in the local type environment, then look its type up there; otherwise, look its type up in the global environment.

RULE


RULE


when $\neg_{Bool}$ X `in keys` ρ

We want the increment operation to apply to any l-value, including array elements, not only to variables. For that reason, we define the following special context which extracts the type of the argument of the increment operation only if that argument is an l-value. Otherwise the rewriting process gets stuck. See the definition of `l-type` at the end of this module. The type of the l-value is expected to be an integer in order to be allowed to be incremented, as seen in the rule "`++ int => int`" below.

CONTEXT: `++` $\dfrac{\square}{\texttt{l-type(}\ \square\ \texttt{)}}$

RULE `++ int` ⇒ `int`

RULE `int + int` ⇒ `int`

RULE `string + string` ⇒ `string`

RULE `int - int` ⇒ `int`

RULE `int * int` ⇒ `int`

RULE `int / int` ⇒ `int`

RULE `int % int` ⇒ `int`

RULE `- int` ⇒ `int`

RULE `int < int` ⇒ `bool`

RULE `int <= int` ⇒ `bool`

RULE `int > int` ⇒ `bool`

RULE `int >= int` ⇒ `bool`

RULE T `==` T ⇒ `bool`

RULE T `!=` T ⇒ `bool`

RULE `bool and bool` ⇒ `bool`

RULE `bool or bool` ⇒ `bool`

RULE `not bool` ⇒ `bool`

RULE `array of` T [`int` , Ts] ⇒ T [Ts]

RULE T [] ⇒ T

RULE `sizeOf( array of` T `)` ⇒ `int`

RULE $Ts \to T$ (Ts) ⇒ T

RULE `read()` ⇒ `int`

The special context and the rule for assignment below are similar to those for the increment operation above: the LHS of the assignment must be an l-value and, in that case, it must have the same type as the RHS, which thus becomes the type of the assignment.

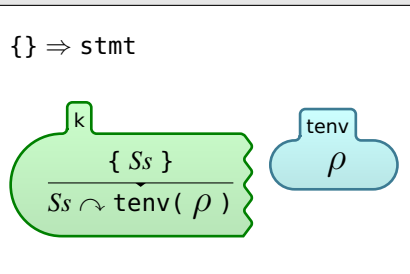
CONTEXT: $\dfrac{\square}{\texttt{l-type(}\ \square\ \texttt{)}}$ = –

RULE T = T ⇒ T

Statements

Statements are also straightforward to be given a typing policy now. Note that the type environment is recovered after each block (see the definition of `tenv` at the end of this module), that the value returned by `return` statements must have the same type as stated in the `return` cell, that the `print` variadic function is allowed to only print integers and strings, and that thrown exceptions can only have integer type.

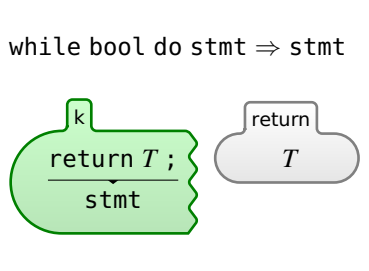
RULE `{}` ⇒ `stmt`

RULE


RULE T ; ⇒ `stmt`

RULE `if bool then stmt else stmt` ⇒ `stmt`

RULE `while bool do stmt` ⇒ `stmt`

RULE


RULE `return;` ⇒ `stmt`

RULE `print(` $\dfrac{\texttt{int}\ ,\ Ts}{Ts}$ `);`

RULE `print(` $\dfrac{\texttt{string}\ ,\ Ts}{Ts}$ `);`

RULE `print();` ⇒ `stmt`

RULE `try stmt catch(` X `)` S ⇒ { `var` X : `int` ; S } [structural]

RULE `throw int ;` ⇒ `stmt`

RULE `spawn stmt` ⇒ `stmt`

RULE `acquire` T ; ⇒ `stmt`

RULE `release` T ; ⇒ `stmt`

RULE `rendezvous` T ; ⇒ `stmt`

RULE `stmt stmt` ⇒ `stmt`

Auxiliary operations

The `l-type` operation below evaluates to the type of its argument, but only if that argument is an l-value, that is, a variable or an array element.

SYNTAX *Exp* ::= `l-type(` *Exp* `)`
RULE `l-type(` X `)` ⇒ X [structural]

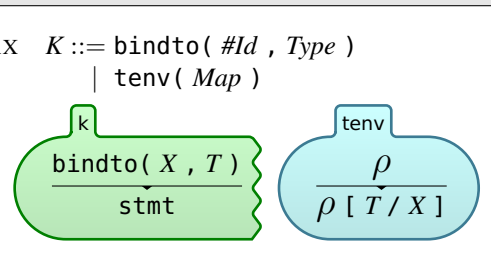
CONTEXT: `l-type(` – [□] `)`

CONTEXT: `l-type(` □ [–] `)`

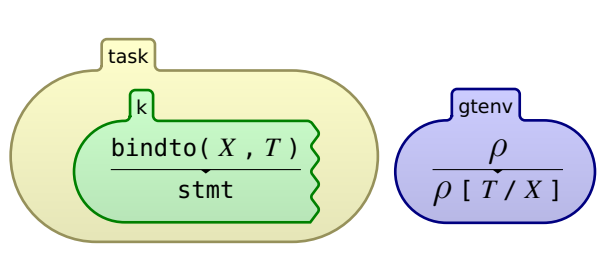
RULE `l-type(` T `)` ⇒ T [structural]

The two operations below are standard, we use them in many $\mathbb{K}$ definitions. Note, however, that `bindto` evaluates to `stmt` and, moreover, that it fist attempts to do the binding locally; if a local environment is not available, meaning that one attempts to bind a top-level variable or function name, then does the binding in the global type environment.

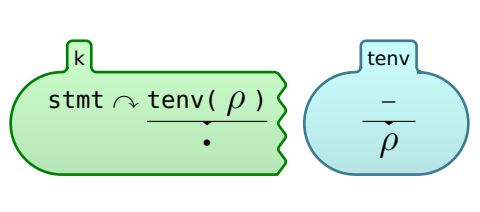
```
SYNTAX  K ::= bindto( #Id , Type )
            | tenv( Map )
```

RULE


[structural]

RULE


[structural]

RULE


[structural]

The operation below makes sure that the list of types passed as its first argument are all integers and there is sufficient "`array of`" depth in the second argument type to justify them. Recall from above that this operation is used to check whether a particular array declaration type-checks.

SYNTAX *K* ::= `checkDepth(` *Types* , *Type* `)`
RULE `checkDepth(` $\dfrac{\texttt{int}\ ,\ Ts}{Ts}$, $\dfrac{\texttt{array of}\ T}{T}$ `)` [structural]

RULE `checkDepth(` • , – `)` ⇒ • [structural]

Finally, the operation below ensures that its argument is a list of typed identifiers (as needed for the parameters in function declarations) and it rewrites to the list of their types.

SYNTAX *Types* ::= `types(` *Exps* `)`
RULE `types()` ⇒ • [structural]

RULE `types(` X : T , XTs `)` ⇒ T , `types(` XTs `)` [structural]

END MODULE