# 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 system/checker for the language.

When designing a type system for a language, no matter in what paradigm, we have to decide upon the intended typing policy. Note that we can have multiple type systems for the same language, one for each 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 for rejecting programs that could perform division-by-zero is not expected to be 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, including array variables. This is done in a C/Java-style. For example, "`int x;`" or "`int x=7, y=x+3;`", or "`int[][][] a[10,20];`" (the latter defines a $10 \times 20$ matrix of arrays of integers). Recall from untyped SIMPLE that, unlike in C/Java, our multi-dimensional arrays use comma-separated arguments, although they have the array-of-array semantics.
- Functions are also typed in a C/Java style. However, since in SIMPLE we allow functions to be passed to and returned by other functions, we also need function types. We will use the conventional higher-order arrow-notation for function types, but will separate the argument types with commas. For example, a function returning an array of `bool` elements and taking as argument an array `x` of two-integer-argument functions returning an integer, is declared using a syntax of the form

  ```
  bool[] f(((int,int)->int)[] x) { ... }
  ```

  and has the type `((int,int)->int)[] -> bool[]`.
- We allow any variable declarations at the top level. Functions can only be declared at the top level. 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 `int->int[][]` and function `g` has the type `int->int`, then the increment expression `++f(7)[g(2),g(3)]` is valid.
- Functions should only return values of their declared result type. To give the programmers more flexibility, 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. Nevertheless, as the dynamic semantics shows, a return value is automatically generated when an explicit `return` statement is not encountered.
- For simplicity, we here limit exceptions to only throw and catch integer values. We let it as an exercise to the reader to extend the semantics to allow throwing and catching arbitrary-type exceptions. Like in programming languages like Java, one can go even further and define a semantics where thrown exceptions are propagated through try-catch statements until one of the corresponding type is found. We will do this when we define the KOOL language, not here. To keep the definition if SIMPLE simple, here we do not attempt to reject programs which throw uncaught exceptions.

Like in untyped SIMPLE, some constructs can be desugared into a smaller set of basic constructs. In general, it should be clear why a program does not type by looking at the top of the textsfk cells in its stuck configuration.

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* ::= $Token\{"main"\}$

### 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* ::= `void`
| `int`
| `bool`
| `string`
| *Type*[]
| *Types* `->` *Type*
| (*Type*) [bracket]

SYNTAX *Types* ::= *List*{*Type*, "," }

### Declarations

Variable and function declarations have the expected syntax. For variables, we basically just replaced the `var` keyword of untyped SIMPLE with a type. For functions, besides replacing the `function` keyword with a type, we also introduce a new syntactic category for typed variables, *Param*, and lists over it.

SYNTAX *Param* ::= *Type* *Id*

SYNTAX *Params* ::= *List*{*Param*, "," }

SYNTAX *Decl* ::= *Type* *Exps* `;`
| *Type* *Id*(*Params*)*Block*

### 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 *Exp* ::= *Int*
| *Bool*
| *String*
| *Id*
| (*Exp*) [bracket]
| `++` *Exp*
| *Exp*[*Exps*] [strict, klabel('_:Exp[_:Exps])]
| *Exp*(*Exps*) [strict]
| `-` *Exp* [strict]
| `sizeOf` (*Exp*) [strict]
| `read` ()
| *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* [strict]
| *Exp* `&&` *Exp* [strict]
| *Exp* `||` *Exp* [strict]
| `spawn` *Block*
| *Exp* `=` *Exp* [strict(2)]

Note that `spawn` has not been declared strict. This may seem unexpected, because the child thread shares the same environment with the parent thread, so from a typing perspective the spawned statement makes the same sense in a child thread as it makes in the parent thread. The reason for not declaring it strict is because we want to disallow programs where the spawned thread calls the `return` statement, because those programs would get stuck in the dynamic semantics. The type semantics of spawn below will reject such programs.

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

SYNTAX *Exps* ::= *List*{*Exp*, "," } [strict]

### Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameter. 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 *Block* ::= {}
| {*Stmts*}

SYNTAX *Stmt* ::= *Decl*
| *Block*
| *Exp* `;` [strict]
| `if` (*Exp*)*Block* `else` *Block* [avoid, strict]
| `if` (*Exp*)*Block*
| `while` (*Exp*)*Block* [strict]
| `for` (*Stmts* *Exp* `;` *Exp*)*Block*
| `return` *Exp* `;` [strict]
| `return ;`
| `print` (*Exps*) `;` [strict]
| `try` *Block* `catch` (*Param*)*Block* [strict(1)]
| `throw` *Exp* `;` [strict]
| `join` *Exp* `;` [strict]
| `acquire` *Exp* `;` [strict]
| `release` *Exp* `;` [strict]
| `rendezvous` *Exp* `;` [strict]

Note that the sequential composition is now sequentially strict, because, unlike in the dynamic semantics where statements dissolved, they now reduce to the `stmt` type, which is a result.

SYNTAX *Stmts* ::= *Stmt*
| *Stmts* *Stmts* [seqstrict]

### Desugaring macros

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

RULE $\dfrac{\texttt{if } (E)S}{\texttt{if } (E)S \texttt{ else } \{\}}$ [macro]

RULE $\dfrac{\texttt{for } (Start\ \ Cond\ \texttt{;}\ Step)\{S{:}Stmts\}}{\{Start\ \ \texttt{while } (Cond)\{S\ \ Step\ \texttt{;}\}\}}$ [macro]

RULE $\dfrac{\texttt{for } (Start\ \ Cond\ \texttt{;}\ Step)\{\}}{\{Start\ \ \texttt{while } (Cond)\{Step\ \texttt{;}\}\}}$ [macro]

RULE $\dfrac{T{:}Type\ \ E1{:}Exp\texttt{,}\ E2{:}Exp\texttt{,}\ Es{:}Exps\ \texttt{;}}{T\ \ E1\ \texttt{;}\ \ T\ \ E2\texttt{,}\ Es\ \texttt{;}}$ [macro]

RULE $\dfrac{T{:}Type\ \ X{:}Id\ \texttt{=}\ E\ \texttt{;}}{T\ \ X\ \texttt{;}\ \ X\ \texttt{=}\ E\ \texttt{;}}$ [macro]

END MODULE

MODULE SIMPLE-TYPED-STATIC

## Static semantics

Here we define the type system of SIMPLE. 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 function 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 reduces 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 did not need types for statements and for blocks as part of the typed SIMPLE syntax, because programmers are not allowed to use such types explicitly. However, we are going to need them in the type system, because blocks and statements reduce to them.

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. Like in the type system of IMP++ in the $\mathbb{K}$ tutorial, we prefer to group the block and statement types under one syntactic sub-category of types, because this allows us to more compactly state that certain terms can be either blocks or statements. Also, since programs and fragments of program will reduce to their types, in order for the strictness and context declarations to be executable we state that types are results (same like we did in the IMP++ tutorial).

SYNTAX *Exp* ::= *Type*

SYNTAX *BlockOrStmtType* ::= `block`
| `stmt`

SYNTAX *Type* ::= *BlockOrStmtType*

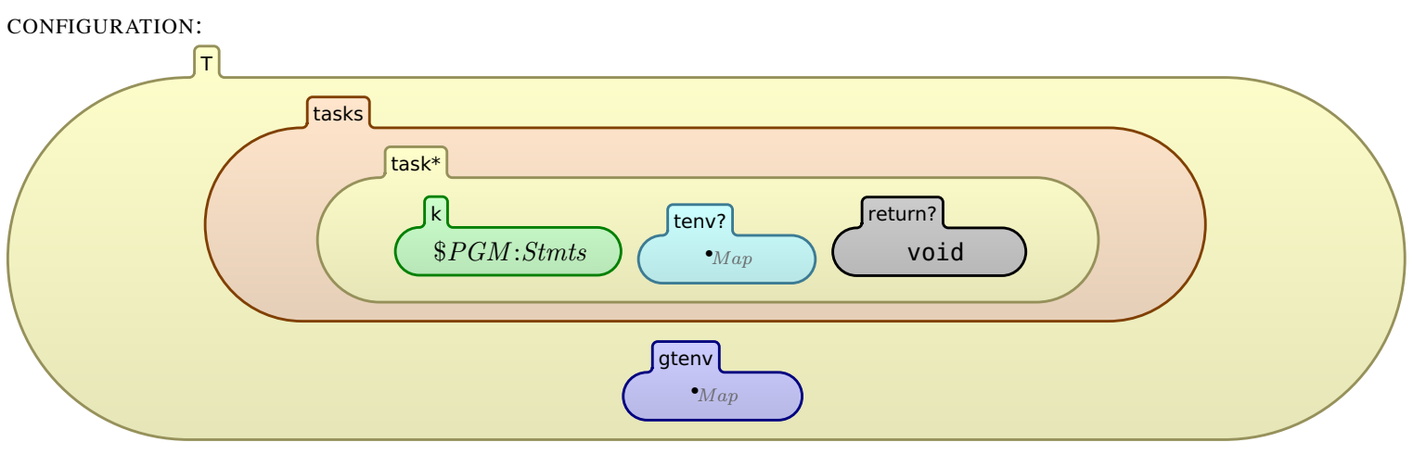
SYNTAX *Block* ::= *BlockOrStmtType*

SYNTAX *KResult* ::= *Type*

### Configuration

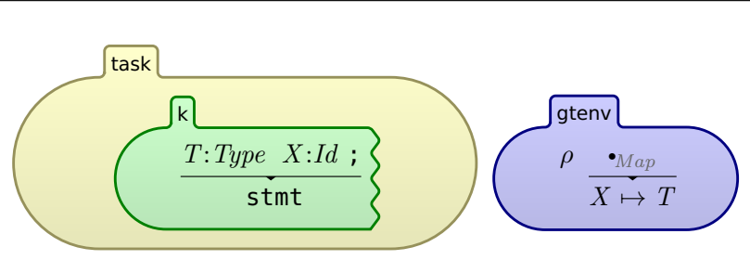
The configuration of our type system consists of a `tasks` cell holding various typing `task` cells, and a global type environment. Each task includes a `k` cell holding the code to type, a `tenv` cell holding the local type environment, and a `return` cell holding the return type of the currently checked function. The latter is needed in order to check whether return statements return values of the expected type. Initially, the program is placed in a `k` cell inside a `task` cell. Since the cells with multiplicity "?" are not included in the initial configuration, the `task` cell holding the original program in its `k` cell will contain no other subcells.

CONFIGURATION:

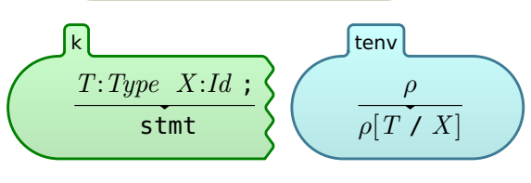


### Variable declarations

Variable declarations type as statements, that is, they reduce to the type `stmt`. There are only two cases that need to be considered: 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 two cases. The first case has two subcases: when the variable declaration is global (i.e., the `task` cell contains only the `k` cell), in which case it is added to the global type environment checking at the same time that the variable has not been already declared; and when the variable declaration is local (i.e., a `tenv` cell is available), in which case it is simply added to the local type environment, possibly shadowing previous homonymous variables. The third case reduces to the second, incrementally moving the array dimension into the type until the array becomes a simple variable.

RULE


requires $\neg_{Bool}(X \texttt{ in keys } (\rho))$

RULE


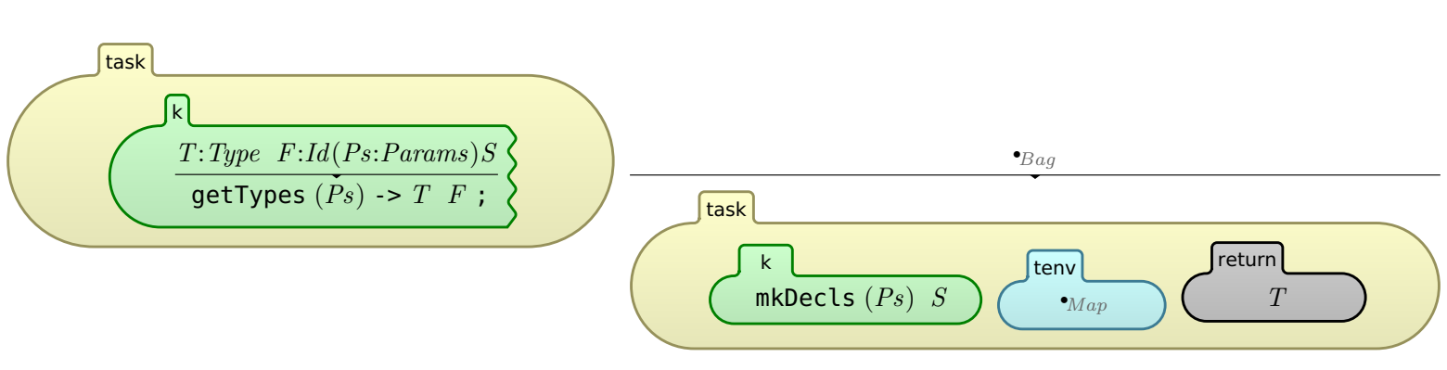
CONTEXT —:Type —:Exp[□] ;

RULE $\dfrac{T{:}Type\ \ E{:}Exp[\ \texttt{int},\ Ts{:}Types]\ \texttt{;}}{T[]\ \ E[Ts]\ \texttt{;}}$ [structural]

RULE $\dfrac{T{:}Type\ \ E{:}Exp[\bullet_{Types}]\ \texttt{;}}{T\ \ E\ \texttt{;}}$ [structural]

### Function declarations

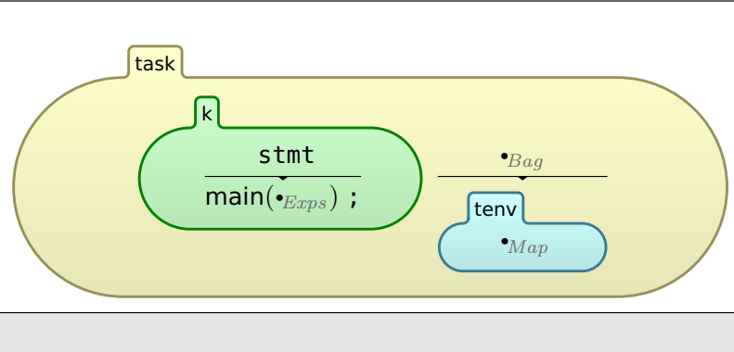
Functions are allowed to be declared only at the top level (the `task` cell holds only its `k` subcell). Each function declaration reduces to a variable declaration (a binding of its name to its declared function type), but also adds a task into the `tasks` cell. The task consists of a typing of the statement declaring all the function parameters followed by the function body, together with the expected return type of the function. The `types` and `mkDecls` functions, defined at the end of the file in the section on auxiliary operations, extracts the list of types and makes a sequence of variable declarations from a list of function parameters, respectively. Note that, although in the dynamic semantics we include a terminating `return` statement at the end of the function body to eliminate from the analysis the case when the function does not provide an explicit return, we do not need to include such a similar `return` statement here. That's because the `return` statements type to `stmt` anyway, and the entire code of the function body needs to type anyway.

RULE


[structural]

### Checking if `main()` exists

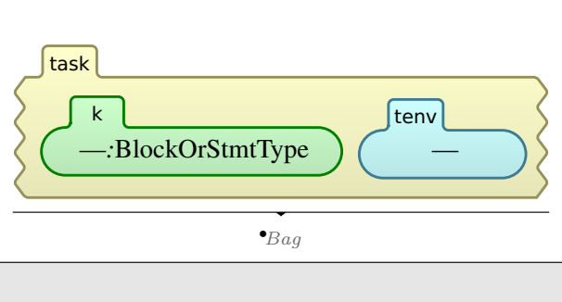
Once the entire program is processed (generating appropriate tasks to type check its function bodies), we can dissolve the main `task` cell (the one holding only a `k` subcell). Since we want to enforce that programs include a main function, we also generate a function task executing `main()` to ensure that it types (remove this task creation if you do not want your type system to reject programs without a `main` function).

RULE


[structural]

### Collecting the terminated tasks

Similarly, once a non-main task (i.e., one which contains a `tenv` subcells) is completed using the subsequent rules (i.e., its `k` cell holds only the `block` or `stmt` type), we can dissolve its corresponding cell. Note that it is important to ensure that we only dissolve tasks containing a `tenv` cell with the rule below, because the main task should *not* dissolve this way! It should do what the above rule says. In the end, there should be no task cell left in the configuration when the program correctly type checks.

RULE


### Basic values

The first three rewrite rules below reduce the primitive values to their types, as we typically do when we define type systems in $\mathbb{K}$.

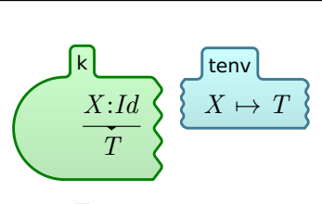
RULE $\dfrac{—{:}Int}{\texttt{int}}$

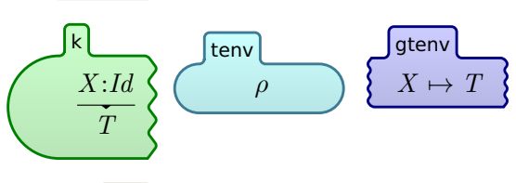
RULE $\dfrac{—{:}Bool}{\texttt{bool}}$

RULE $\dfrac{—{:}String}{\texttt{string}}$

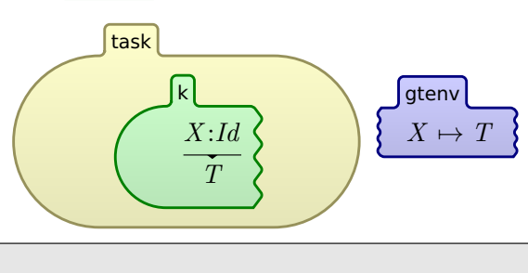
### Variable lookup

There are three cases to distinguish for variable lookup: (1) if the variable is bound in the local type environment, then look its type up there; (2) if a local environment exists and the variable is not bound in it, then look its type up in the global environment; (3) finally, if there is no local environment, meaning that we are executing the top-level pass, then look the variable's type up in the global environment, too.

RULE


RULE


requires $\neg_{Bool}(X \texttt{ in keys } (\rho))$

RULE


### Increment

We want the increment operation to apply to any lvalue, including array elements, not only to variables. For that reason, we define a special context extracting the type of the argument of the increment operation only if that argument is an lvalue. Otherwise the rewriting process gets stuck. The operation `ltype` is defined at the end of this file, in the auxiliary operation section. It essentially acts as a filter, getting stuck if its argument is not an lvalue and letting it reduce otherwise. The type of the lvalue is expected to be an integer in order to be allowed to be incremented, as seen in the rule "`++ int => int`" below.

CONTEXT $\texttt{++}\ \dfrac{\square}{\texttt{ltype}\ (\square)}$

RULE $\dfrac{\texttt{++ int}}{\texttt{int}}$

### Common expression constructs

The rules below are straightforward and self-explanatory:

RULE $\dfrac{\texttt{int + int}}{\texttt{int}}$

RULE $\dfrac{\texttt{string + string}}{\texttt{string}}$

RULE $\dfrac{\texttt{int - int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int * int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int / int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int \% int}}{\texttt{int}}$

RULE $\dfrac{\texttt{- int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int < int}}{\texttt{bool}}$

RULE $\dfrac{\texttt{int <= int}}{\texttt{bool}}$

RULE $\dfrac{\texttt{int > int}}{\texttt{bool}}$

RULE $\dfrac{\texttt{int >= int}}{\texttt{bool}}$

RULE $\dfrac{T{:}Type\ \texttt{==}\ T}{\texttt{bool}}$

RULE $\dfrac{T{:}Type\ \texttt{!=}\ T}{\texttt{bool}}$

RULE $\dfrac{\texttt{bool \&\& bool}}{\texttt{bool}}$

RULE $\dfrac{\texttt{bool || bool}}{\texttt{bool}}$

RULE $\dfrac{\texttt{! bool}}{\texttt{bool}}$

### Array access and size

Array access requires each index to type to an integer, and the array type to be at least as deep as the number of indexes:

RULE $\dfrac{(T[])[\ \texttt{int},\ Ts{:}Types]}{T[Ts]}$

RULE $\dfrac{T{:}Type[\bullet_{Types}]}{T}$

`sizeOf` only needs to check that its argument is an array:

RULE $\dfrac{\texttt{sizeOf}\ (T[])}{\texttt{int}}$

### 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.

RULE $\dfrac{\texttt{read}\ ()}{\texttt{int}}$

RULE $\texttt{print}\ (\dfrac{T{:}Type,\ Ts}{Ts})\ \texttt{;}$ requires $T =_K \texttt{int} \vee_{Bool} T =_K \texttt{string}$

RULE $\dfrac{\texttt{print}\ (\bullet_{Types})\ \texttt{;}}{\texttt{stmt}}$

### Assignment

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

CONTEXT $\dfrac{\square}{\texttt{ltype}\ (\square)}\ \texttt{=}\ —$

RULE $\dfrac{T{:}Type\ \texttt{=}\ T}{T}$

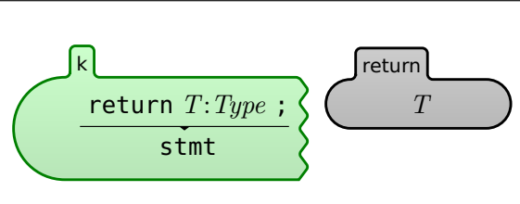
### Function application and return

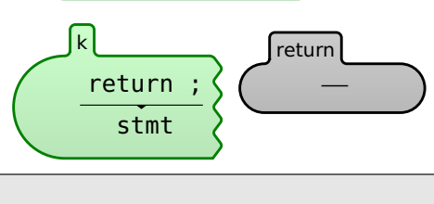
Function application requires the type of the function and the types of the passed values to be compatible. Note that a special case is needed to handle the no-argument case:

RULE $\dfrac{(Ts{:}Types\ \texttt{->}\ T)(Ts)}{T}$ requires $Ts \neq_K \bullet_{Types}$

RULE $\dfrac{(\ \texttt{void ->}\ T)(\bullet_{Types})}{T}$

The returned value must have the same type as the declared function return type. If an empty return is encountered, than we should check that we are in a function (and not a thread) context, that is, a `return` cell must be available:

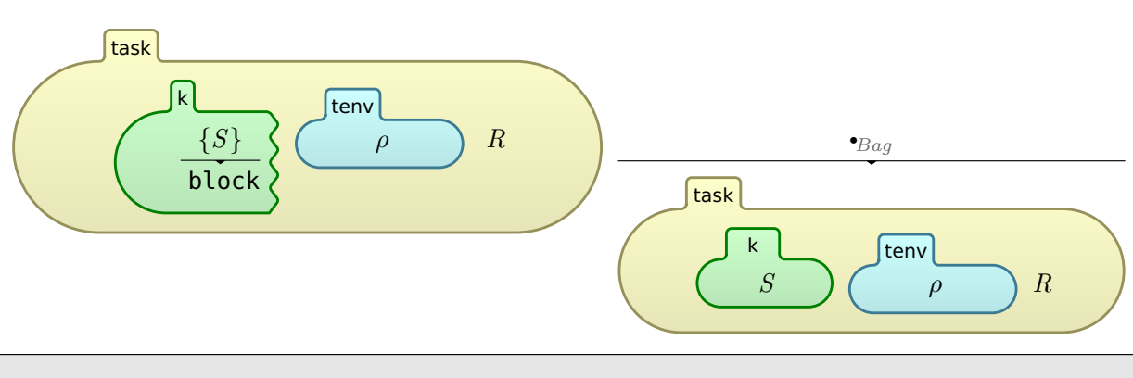
RULE


RULE


### Blocks

To avoid having to recover type environments after blocks, we prefer to start a new task for block body, making sure that the new task is passed the same type environment and return cells. The value returned by `return` statements must have the same type as stated in the `return` cell. The `print` variadic function is allowed to only print integers and strings. The thrown exceptions can only have integer type.

RULE $\dfrac{\{\}}{\texttt{block}}$

RULE


### Expression statement

RULE $\dfrac{—{:}Type\ \texttt{;}}{\texttt{stmt}}$

### Conditional and `while` loop

RULE $\dfrac{\texttt{if}\ (\ \texttt{bool})\ \texttt{block else block}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{while}\ (\ \texttt{bool})\ \texttt{block}}{\texttt{stmt}}$

### Exceptions

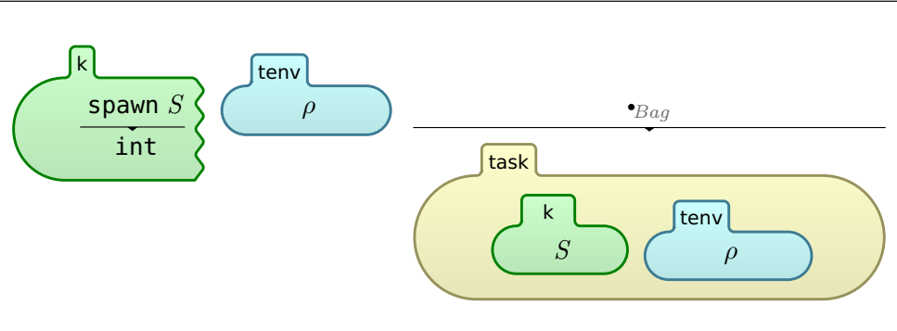
We currently force the parameters of exceptions to only be integers. Moreover, for simplicity, we assume that integer exceptions can be thrown from anywhere, including from functions which do not define any try-catch block (with the currently unchecked—also for simplicity—expectation that the caller functions would catch those exceptions).

RULE $\dfrac{\texttt{try block catch}\ (\ \texttt{int}\ \ X{:}Id)\{S\}}{\{\ \texttt{int}\ \ X\ \texttt{;}\ \ S\}}$ [structural]

RULE $\dfrac{\texttt{throw int ;}}{\texttt{stmt}}$

### Concurrency

Nothing special about typing the concurrency constructs, except that we do not want the spawned thread to return, so we do not include any `return` cell in the new task cell for the thread statement. Same like with the functions above, we do not check for thrown exceptions which are not caught.

RULE


RULE $\dfrac{\texttt{join int ;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{acquire}\ —{:}Type\ \texttt{;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{release}\ —{:}Type\ \texttt{;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{rendezvous}\ —{:}Type\ \texttt{;}}{\texttt{stmt}}$

RULE $\dfrac{—{:}BlockOrStmtType\ \ —{:}BlockOrStmtType}{\texttt{stmt}}$

## Auxiliary constructs

The function `mkDecls` turns a list of parameters into a list of variable declarations.

SYNTAX *Decl* ::= `mkDecls` (*Params*) [function]

RULE $\dfrac{\texttt{mkDecls}\ (T{:}Type\ \ X{:}Id\texttt{,}\ Ps{:}Params)}{T\ \ X\ \texttt{;}\ \ \texttt{mkDecls}\ (Ps)}$

RULE $\dfrac{\texttt{mkDecls}\ (\bullet_{Params})}{\{\}}$

The `ltype` context allows only expressions which have an lvalue to evaluate.

SYNTAX *LValue* ::= *Id*

RULE $\dfrac{isLValue(—{:}Exp[—{:}Exps])}{\texttt{true}}$

SYNTAX *Exp* ::= `ltype` (*Exp*)

CONTEXT `ltype` (□) requires $isLValue(\square)$

The function `getTypes` is the same as in SIMPLE typed dynamic.

SYNTAX *Types* ::= `getTypes` (*Params*) [function]

RULE $\dfrac{\texttt{getTypes}\ (T{:}Type\ \ —{:}Id)}{T\ \texttt{,}\ \bullet_{Types}}$

RULE $\dfrac{\texttt{getTypes}\ (T{:}Type\ \ —{:}Id\texttt{,}\ P\texttt{,}\ Ps)}{T\ \texttt{,}\ \texttt{getTypes}\ (P\texttt{,}\ Ps)}$

RULE $\dfrac{\texttt{getTypes}\ (\bullet_{Params})}{\texttt{void,}\ \bullet_{Types}}$

END MODULE