SIMPLE — Typed — Dynamic

Grigoriou Rupa and Tsiaze Florin Jaybeejii (grprss,tsorhan2)@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

This is an 18 segment recreation of the typed SIMPLE language. It is very similar to the semantics of the untyped SIMPLE, the differences being that we now dynamically check the typing policy described in the static semantics of typed SIMPLE. Additionally, we have added a new type, `nothing`, which is the bottom type and is the type of expressions that never return successfully, such as erroneous blocks (as in accessing an array out of its 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` and `Static SIMPLE` documents for more comments on the typed SIMPLE language.

BRIEFLY: 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 `X` universe attributes are like those of the untyped SIMPLE, to support the desired evaluation strategies of the various language constructs.

```
universe Id := macro
```

Types

```
inductive Type :=
  | void
  | int
  | bool
  | char
  | Pair : * -> * -> Type
  | Type-Literal X

inductive Types := List(Type, * )
```

Declarations

```
inductive Formal := Type Id

inductive Process := List(Process, * )

inductive Decl := Type Exp |
                  Type Id(Formals)Block
```

Expressions

```
inductive Exp :=
  | Id
  | Num
  | Bool
  | Char
  | Binop(Exp, Exp)
  | Not(Exp)
  | Neg(Exp)
  | Deref(Exp)
  | Addr(Exp)
  | Index(Exp, Exp)
  | Call(Exp, Exp)
  | Field(Exp, Id)
  | Cast(Type, Exp)
  | Type-Literal X
  | Pair(Exp, Exp)
  | Type-Literal(X)
  | Alloc(Exp)
  | Size(Exp)
  | Read(Exp)

inductive Exps := List(Exp, * )
```

Like in the static semantics, there is the need for lists of identifiers (because we now have lists of parameters).

```
inductive Ids := List(Id, * )
```

Statements

```
inductive Block := Ss
                 | Error

inductive Stmt := Decl
                | Exp
                | Print(Exp)
                | If_Else(Exp,Block,Block)[scope,nest(2)]
                | While(Exp,Block)
                | Write(Exp,Exp)
                | Return(Exp)[nest(2)]
                | Break(Exp)[nest(2)]
                | Continue(Exp)[nest(2)]
                | Exp:=Exp[nest(2)]
                | Exp+=Exp[nest(2)]
                | Exp-=Exp[nest(2)]
                | Exp*=Exp[nest(2)]
                | Exp/=Exp[nest(2)]
                | Exp%=Exp[nest(2)]
                | Exp++[nest(2)]
                | Exp--[nest(2)]
                | Type-Literal X
                | Type-Literal(X)[nest(2)]
                | Assert Bool
                | Assert(Bool, Exp)[nest(2)]

inductive Stmts := Ss
                 | Stmt Stmts
```

The same decoupling done to the file statically typed SIMPLE.

```
universe id_or_d
  := id_or_d Id
   | Pair(Pair(Id, Id), Pair(Id, Id))
   | Pair(id_or_d, id_or_d) | Proj (X)

universe d_or_e
  := d_or_e Decl
   | d_or_e e

universe d_or_e_
  := d_or_e_ d
   | d_or_e_ e
```

[macro]

[macro]

[macro]

[macro]

BRIEFLY: 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 their types. These types are needed in order to easily compute the type of any value in the language (not the auxiliary `typeOf` operator at the end of the section).

```
inductive Val := Int
               | Bool
               | Array(Type, Nat, Int)
               | Lambda (Type,Params, Stmt)

inductive Vals := List(Val, * )

inductive Flow := Val

inductive Effect := Val
```

Configuration

The configuration is almost identical to that of untyped SIMPLE, except for a return call inside the control cell. The return call will hold, like in the static semantics of typed SIMPLE, the expected type of the value returned by the function being called. Similarly to the static semantics, the return call is created and will be checked whenever the control cell becomes of the function body encounters an explicit return statement.

CONFIGURATION:

Declaration and Initialization

Variable Declaration. The "hardwired" construct is now parameterized by a type. A main difference between untyped SIMPLE and typed SIMPLE is that in typed SIMPLE, the type of a variable is fixed at the time of its declaration and it cannot be changed during the execution of the program. We do not do any statistical, transpositional or any semantic definitions here, as we have done in the static semantics of typed SIMPLE. Variables are still allocated in the store, and they are still initialized to 0 or an error element, whose type was explicitly or implicitly declared in the program and does not change. It is therefore safe to assume that the type of a variable is the type of the last cell in a binage. The typed attribute values effectively accept both a type and an initialized value in a location.

```
universe M := Id
            | Num
```


Array Declaration. The dynamic semantics of typed array declarations is similar to that in untyped SIMPLE, but we have to make sure that we associate the right type to the allocated elements.

Function declaration. Store all function parameters, as well as the return type, as part of the lambda abstraction. As the scope of dynamic typing, we will make sure that parameters are not declared more than once in the same block.

Calling main()

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

```
universe M := execute
            | Id
```


Expressions

Variable lookup.

Variable/Array increment.

```
universe exp
  := lvalue(X)
   | rvalue(X)
```


Arithmetic operators.

```
universe M := Id
            | Num
            | Bool(Bool)
            | Bool_Binop_Bool
            | Num_Binop_Num
            | Num_Binop_Bool
            | Id = Id
            | Id = Num
            | Id = Bool
            | Id = Char
            | Id = Id Binop Id
            | Id = Num Binop Num
            | Id = Id Binop Num
            | Id = Num Binop Id
            | Id += Id
            | Id += Num
            | Id -= Id
            | Id -= Num
            | Id *= Id
            | Id *= Num
            | Id /= Id
            | Id /= Num
            | Id %= Id
            | Id %= Num
            | Id++
            | Id--
            | Cast(Id, X)
            | Id = Cast(Id, X)
            | Id = Id
            | Num
            | Bool
            | Id = Id
            | Num
```

Array Setup. Check array bounds, as part of the dynamic typing policy.

Size of an array.

Function call. Define functions call and return together, as one their relationship. Note that this operation **MERGES** into declare at parallel, typed assignment modalities, and that the semantics of return also checks that the type of the returned value is correct.

Like the untyped SIMPLE above, modifying also gets tagged with a type now. The empty `return` statement is completed to return the `nothing` value tagged as expected.

```
universe Val := nothing(Type)
```

Read.

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

Statements

Blocks.

Sequential composition.

Expression statements.

```
  | Exp
  | Exp
```

Conditionals.

While loop.

Print. We only allow printing integers and strings.

Exceptions. Exception parameters are now typed, but note that the semantics below works correctly only when the dynamic type of the exception is the same as the declared type of the exception. Also, note that the `Error` block is the same as in the untyped and dynamic and catches integer values, in which case our semantics below works fine.

Threads.

Thread creation.

Thread termination

Thread joining

Acquire lock

Release lock

Readers/writer synchronization

Auxiliary declarations and operations

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

```
universe Decl := mkDecls(Formals,Vals) [Section]

universe mkDecls(X,Y) := mkDecls(X,Y,[],[])
                       | mkDecls(X,[],Ys,[])
                       | mkDecls([],Xs,[],Ys)
                       | mkDecls([],[],[],[])
```

Location lookup.

```
universe M := lookup_Int
            | lookup_Bool
            | lookup_Char
```


Environment extension.

Value and Ivs ...

```
universe Exp := Value(Id)
universe Val := Ivs_Int
```


Adds the corresponding depth to an array type

```
universe Type := Dim * Nat | Checked
```

Sequences of locations.

```
universe Map := Nat, Int -> Checked
```



```
universe Ids := Id
              | Ids, Id

universe Val := TypeOf Id [Checked]
```

```
  | typeOf(Id)
  | typeOf(Num)
  | typeOf(Bool)
  | typeOf(Char)
  | typeOf(Id Binop Id)
  | typeOf(Num Binop Num)
  | typeOf(Id Binop Num)
  | typeOf(Num Binop Id)
  | typeOf(Not(Id))
  | typeOf(Neg(Id))
  | typeOf(Deref(Id))
  | typeOf(Addr(Id))
  | typeOf(Index(Id, Id))
  | typeOf(Call(Id, Ids))
  | typeOf(Field(Id, Id))
  | typeOf(Cast(Type, Id))
  | typeOf(Pair(Id, Id))
  | typeOf(Alloc(Id))
  | typeOf(Size(Id))
  | typeOf(Read(Id))
```

List of types of a parameter

```
universe Types := getTypes(Formals) [Section]

universe getTypes(X) := X
                      | getTypes([])
                      | getTypes(Id, Fs)
                      | getTypes(Id, [])
                      | getTypes(Id, Id, Fs)
                      | getTypes(Id, Id, [])
                      | getTypes(Id, Id, Id, Fs)
                      | getTypes(Id, Id, Id, [])
                      | getTypes(Id, Id, Id, Id, Fs)
                      | getTypes(Id, Id, Id, Id, [])
```