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 check the typing policy (described in the static semantics of typed SIMPLE) dynamically. Because of the dynamic nature of the semantics, we can also perform some additional checks which were not possible in the static semantics, such as memory leaks due to 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.

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 strictness attributes are like those of the untyped SIMPLE, to capture the desired evaluation strategies of the various language constructs .

SYNTAX *#Id* ::= `main`

Types

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

Declarations

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

Expressions

```
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

```
SYNTAX Stmt ::= {}
              | { Stmts }
              | Exp ; [strict]
              | if Exp then Stmt else Stmt [strict(1)]
              | if Exp then Stmt
              | while Exp do Stmt
              | for #Id = Exp to Exp do Stmt
              | return Exp ; [strict]
              | return;
              | print( Exps ); [strict]
              | try Stmt catch( Exp ) Stmt
              | throw Exp ; [strict]
              | spawn Stmt
              | acquire Exp ; [strict]
              | release Exp ; [strict]
              | rendezvous Exp ; [strict]
SYNTAX Stmts ::= Decl
               | Stmt
               | Stmts Stmts
```

We use the same desugaring macros like in typed SIMPLE

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-DYNAMIC
IMPORTS SIMPLE-TYPED-DYNAMIC-SYNTAX
SYNTAX *Vals* ::= *List{Val,","}*

Semantics

Values and results

These are similar to those of untyped SIMPLE, except that the array references and the function abstrations now also hold their types. These types are needed in order to easily compute the type of any value inthe language (see the auxiliary `typeOf` operation at the end of this module). Note that our more-generous-than-needed syntax here allows function abstractions to take a list of expressions as parameters; in fact, the semantics will be given in a way that those expressions can only be typed identifiers. Recall that the purpose of syntax in a 𝕂 definition is not to parse programs in order to reject those not satisfying the expected syntactic/typing conventions (this is what a static semantics does—see the statically typed SIMPLE), but only to parse programs "enough" to give them semantics. In other words, the 𝕂 syntax is the "syntax of the semantics".

```
SYNTAX Val ::= #Int
             | #Bool
             | #String
             | array( Type , #Nat , #Nat )
             | (λExps . Stmt) : Type
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. This `return` cell will hold, like in the static semantics of typed SIMPLE, the expected type of the value returned by the function being executed. The contents of this cell will be set whenever a function is invoked and will be checked whenever the evaluation of the function body encounters an explicit `return` statement.

CONFIGURATION:

T: threads: thread*: [k: $PGM ↷ execute | control: [fstack: • | xstack: • | return: •] | env: • | holds: •] | genv: • | store: • | busy: • | in: • | out: • | nextLoc: 0

Declarations and Initializations

The "undefined" construct is now parameterized by a type. A main difference between untyped SIMPLE and dynamically typed SIMPLE is that the latter assigns a type to each of its locations and that type cannot be changed during the execution of the program. We do not do any memory management in our semantic definitions here, so locations cannot be reclaimed, garbage collected and/or reused. Each location corresponds precisely to an allocated variable or array element, whose type was explicitly or implicitly declared in the program and does not change. It is therefore safe to type each location and then never allow that type to change. The typed undefined values effectively assign both a type and an undefined value to a location.

SYNTAX $K ::= \bot_{Type}$

RULE k: `var` X `:` T `;` ⇒ • | env: Env ⇒ Env [L / X] | store: • ⇒ $L \mapsto \bot_T$ | nextLoc: L ⇒ $L +_{Nat} 1$

The dynamic semantics of typed array declarations is similar to that in untyped SIMPLE, but we have to enforce that the type of an array declaration be of the form `array of` T and assign the right type (T) to the allocated array locations.

RULE k: `var` X `[` N `] : array of` T `;` ⇒ • | env: Env ⇒ Env [L / X] | store: • ⇒ $L \mapsto$ `array( array of` T `,` $L +_{Nat} 1$ `,` N `)` $L +_{Nat} 1$ `..` $N +_{Nat} L \mapsto \bot_T$ | nextLoc: L ⇒ $N +_{Nat} L +_{Nat} 1$

CONTEXT: `var – [ □ ] : – ;`

The desugaring of multi-dimensional arrays into unidimensional ones is also similar to that in untyped SIMPLE, although we have to make sure that all the declared temporary variables have the right types.

```
SYNTAX #Id ::= $1
             | $2
```

RULE `var` X `[` N_1 `,` N_2 `,` Vs `] : array of array of` T `;` ⇒ `var` X `[` N_1 `] : array of array of` T `; { var $1 : array of array of` T `=` X `; for $2 = 0 to` N_1 `- 1 do { var` X `[` N_2 `,` Vs `] : array of` T `; $1 [ $2 ] =` X `; } }` [structural]

Store all function parameters, as well as the return type, as part of the lambda abstraction. In the spirit of dynamic typing, we will make sure that parameters are well typed when the function is invoked.

RULE k: `function` F `(` XTs `) :` T S ⇒ • | env: Env ⇒ Env [L / F] | store: • ⇒ $L \mapsto (\lambda XTs . S)$ `: types(` XTs `)` $\to T$ | nextLoc: L ⇒ $L +_{Nat} 1$

Calling `main()`

When done with the first pass, call `main()`.

SYNTAX K ::= `execute`

RULE k: `execute` ⇒ `main () ;` | env: Env | genv: • ⇒ Env

Expressions and statements

RULE k: X ⇒ V | env: $X \mapsto L$ | store: $L \mapsto V$ [transition]

CONTEXT: `++` □ ⇒ `l-value( □ )`

RULE k: `++ loc(` L `)` ⇒ $I +_{Int} 1$ | store: $L \mapsto I$ ⇒ $I +_{Int} 1$ [transition]

RULE I_1 `+` I_2 ⇒ $I_1 +_{Int} I_2$

RULE Str_1 `+` Str_2 ⇒ $Str_1 +_{String} Str_2$

RULE I_1 `-` I_2 ⇒ $I_1 -_{Int} I_2$

RULE I_1 `*` I_2 ⇒ $I_1 *_{Int} I_2$

RULE I_1 `/` I_2 ⇒ $I_1 \div_{Int} I_2$ when $I_2 \neq_{Bool} 0$

RULE I_1 `%` I_2 ⇒ $I_1 \%_{Int} I_2$ when $I_2 \neq_{Bool} 0$

RULE `-` I ⇒ $-_{Int} I$

RULE I_1 `<` I_2 ⇒ $I_1 <_{Int} I_2$

RULE I_1 `<=` I_2 ⇒ $I_1 \leq_{Int} I_2$

RULE I_1 `>` I_2 ⇒ $I_1 >_{Int} I_2$

RULE I_1 `>=` I_2 ⇒ $I_1 \geq_{Int} I_2$

RULE V_1 `==` V_2 ⇒ $V_1 =_{Bool} V_2$

RULE V_1 `!=` V_2 ⇒ $V_1 \neq_{Bool} V_2$

RULE B_1 `and` B_2 ⇒ $B_1 \wedge_{Bool} B_2$

RULE B_1 `or` B_2 ⇒ $B_1 \vee_{Bool} B_2$

RULE `not` B ⇒ $\neg_{Bool} B$

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

RULE V `[` N_1 `,` N_2 `,` Vs `]` ⇒ V `[` N_1 `] [` N_2 `,` Vs `]` [structural]

RULE `array( – ,` L `,` M `) [` N `]` ⇒ `lookup(` $N +_{Int} L$ `)` when $N <_{Nat} M \wedge_{Bool} N \geq_{Nat} 0$ [structural]

RULE `sizeOf( array( – , – ,` N `) )` ⇒ N

Define function call and return together, to see their relationship.

SYNTAX *ListItem* ::= `(` *Map* `,` *K* `,` *Bag* `)`

RULE k: $(\lambda XTs . S)$ `: – →` T `(` Vs `)` ↷ K ⇒ `bindto(` XTs `,` Vs `)` ↷ S ↷ `return;` | control: [fstack: • ⇒ `(` Env `,` K `,` return: T' C `)` | return: T' ⇒ T | C] | env: Env ⇒ $GEnv$ | genv: $GEnv$

RULE k: `return` V `;` ↷ – ⇒ V ↷ K | control: [fstack: `(` Env `,` K `,` C `)` ⇒ • | return: T – ⇒ C] | env: – ⇒ Env when `typeOf(` V `)` $=_{Bool} T$

SYNTAX *Val* ::= `nothing`

RULE `return;` ⇒ `return nothing ;` [structural]

RULE k: `read()` ⇒ I | in: I ⇒ • [transition]

CONTEXT: □ `= –` ⇒ `l-value( □ )`

RULE k: `loc(` L `) =` V ⇒ V | store: $L \mapsto K$ ⇒ V when `typeOf(` V `)` $=_{Bool}$ `typeOf(` K `)` [transition]

RULE `{}` ⇒ • [structural]

RULE k: `{` S `}` ⇒ S ↷ `env(` Env `)` | env: Env [structural]

RULE S_1 S_2 ⇒ S_1 ↷ S_2 [structural]

RULE V `;` ⇒ •

RULE `if true then` S `else –` ⇒ S

RULE `if false then – else` S ⇒ S

RULE k: `while` B `do` S ⇒ `if` B `then {` S `while` B `do` S `} else {}` [structural]

We only allow printing integers and strings

RULE k: `print(` V `,` Es `);` ⇒ `print(` Es `);` | out: • ⇒ V when `typeOf(` V `)` $=_{Bool}$ `int` $\vee_{Bool}$ `typeOf(` V `)` $=_{Bool}$ `string` [transition]

RULE `print();` ⇒ • [structural]

SYNTAX *ListItem* ::= `(` *Exp* `,` *Stmt* `,` *K* `,` *Map* `,` *Bag* `)`

SYNTAX K ::= `popx`

RULE k: `try` S_1 `catch(` X `:` T `)` S_2 ↷ K ⇒ S_1 ↷ `popx` | control: [xstack: • ⇒ `(` X `:` T `,` S_2 `,` K `,` Env `,` C `)` | C] | env: Env

RULE k: `popx` ⇒ • | xstack: `( – )` ⇒ •

RULE k: `throw` V `;` ↷ – ⇒ `{ var` X `:` T `=` V `;` S_2 `}` ↷ K | control: [xstack: `(` X `:` T `,` S_2 `,` K `,` Env `,` C `)` ⇒ • | – ⇒ C] | env: – ⇒ Env when `typeOf(` V `)` $=_{Bool} T$

RULE thread: [k: `spawn` S ⇒ • | env: Env] | • ⇒ thread: [k: S | env: Env]

RULE thread: [k: • | holds: H] ⇒ • | busy: $Busy$ ⇒ $Busy$ `keys` H

RULE k: `acquire` V `;` ⇒ • | holds: • ⇒ $V \mapsto 0$ | busy: $Busy$ ⇒ $Busy$ • ⇒ V when $\neg V$ `in` $Busy$

RULE k: `acquire` V `;` ⇒ • | holds: $V \mapsto N$ ⇒ $N +_{Nat} 1$

RULE k: `release` V `;` ⇒ • | holds: $V \mapsto N$ ⇒ $N -_{Nat} 1$ when $N >_{Nat} 0$

RULE k: `release` V `;` ⇒ • | holds: $V \mapsto 0$ ⇒ • | busy: V ⇒ •

RULE k: `rendezvous` V `;` ⇒ • | k: `rendezvous` V `;` ⇒ •

Auxiliary operations

l-value

SYNTAX *Exp* ::= `l-value(` *K* `)`

SYNTAX *Val* ::= `loc(` *#Nat* `)`

RULE k: `l-value(` X `)` ⇒ `loc(` L `)` | env: $X \mapsto L$

CONTEXT: `l-value( □ [ – ] )`

RULE `l-value( lookup(` L `) )` ⇒ `loc(` L `)` [structural]

lookup

SYNTAX K ::= `lookup(` *#Nat* `)`

RULE k: `lookup(` L `)` ⇒ V | store: $L \mapsto V$ [transition]

bindto

`bindto` also checks the well-formedness of the function parameters

SYNTAX K ::= `bindto(` *Exps* `,` *Vals* `)`

RULE k: `bindto(` X `:` T `,` XTs `,` V `,` Vs `)` ⇒ `bindto(` XTs `,` Vs `)` | env: Env ⇒ Env [L / X] | store: • ⇒ $L \mapsto V$ | nextLoc: L ⇒ $L +_{Nat} 1$ when `typeOf(` V `)` $=_{Bool} T$

RULE `bindto( • , • )` ⇒ • [structural]

env

SYNTAX K ::= `env(` *Map* `)`

RULE k: `env(` Env `)` ⇒ • | env: – ⇒ Env [structural]

RULE `env( – )` ↷ `env( – )` ⇒ `env( – )` [structural]

typeOf

SYNTAX *Type* ::= `typeOf(` *K* `)`

RULE `typeOf(` I `)` ⇒ `int` [structural]

RULE `typeOf(` B `)` ⇒ `bool` [structural]

RULE `typeOf( – )` ⇒ `string` [structural]

RULE `typeOf( array(` T `, – , – ) )` ⇒ T [structural]

RULE `typeOf( ( – ) :` T `)` ⇒ T [structural]

RULE `typeOf(` $\bot_T$ `)` ⇒ T [structural]

RULE `typeOf( nothing )` ⇒ `void` [structural]

SYNTAX *Types* ::= `types(` *Exps* `)`

RULE `types( • )` ⇒ • [structural]

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

Sequences of locations

SYNTAX *List{K}* ::= *#Nat* `..` *#Nat*

RULE N_1 `..` N_1 ⇒ •

RULE N_1 `..` N_2 ⇒ N_1 `,` $N_1 +_{Nat} 1$ `..` N_2 when $N_1 <_{Nat} N_2$

END MODULE