Soft Typing for JavaScript

Liewe Thomas van Binsbergen João Paulo Pizani Flor
Pepijn Kokke

June 3rd, 2013

Parsing JavaScript

Using `Language.JavaScript`[1]

We started out using the `Language.JavaScript` package, after it was approved with the remark that with all the clutter of lexical information the AST made the parser look more like a lexer than an actual parser.

Sadly, this turned out to be closer to the truth than we originally imagined. After the creation of a module that removed all the lexical clutter from the AST[2], the tree we ended up with contained many structures such as the following.

```
data JSNode = ...
            | JSExpression [JSNode] -- contains operator tokens,
            | ...                   -- expressions, etc
```

These structures would require a parser of their own to interpret into sensible expressions—an issue that will only show once you try to use the JavaScript AST. However, since the objective of `Language.JavaScript` was to create a valid JavaScript *parser* in Haskell, this was probably never noticed.

For this reason we decided to move to a parser that, while it hasn't been worked on in quite some time, was actually used to write a JavaScript interpreter.

Using the `HJS` Parser[3]

The `HJS` parser—as opposed to `Language.JavaScript`—*does* group expressions according to their semantics. This means, for instance, that the assignment

[1] See http://hackage.haskell.org/package/language-javascript-0.5.7.
[2] This module is no longer part of the package, but may be found in the histories of our git commit logs.
[3] See http://hackage.haskell.org/package/hjs-0.2.1.

operators are no longer represented as generic binary operators, but as a different class of statements.

This made the interpretation of the `HJS` AST much, much simpler, and in mere hours we were able to make more progress that we had made in the weeks before.

However, the semantic separation that `HJS` imposes causes the AST to become much more complex, as it makes distinctions such as effectful and non-effectful, prefix or postfix, etc...

We were able to convert a large number of these expressions to our analysis language: `Simpl`.[4]

The `Simpl` Language

The `Simpl` language is defined as follows. The two basic type definitions are identifiers and labels—which are used to identify expressions in the analysis framework.

```
type Ident = String
type Label = Int
```

The most primitve expressions then are `Atom`s which, as in JavaScript, can be numbers, strings, regular expressions, booleans, identifiers—for variables—and the two values `null` and `undefined`.

```
data Atom = IntegerLit Int
          | StringLit String
          | RegexLit String String
          | BoolLit Bool
          | Ident Ident
          | NullLit
          | UndefinedLit
```

Then we have expressions, which can represent all JavaScript's operators, and atomic expressions.

```
data Exp
  -- * arithmetic expressions
  = Add  Exp Exp | Sub  Exp Exp
  | Mul  Exp Exp | Div  Exp Exp
  | Mod  Exp Exp | Neg  Exp
```

[4] The implementation of the `HJS` to `Simpl` conversion algorithm can be found in the module `Language.Javascript.Hjs2Simpl`.

```
  -- * boolean expressions
  | And  Exp Exp | Or   Exp Exp
  | Not  Exp     | Ter  Exp Exp Exp
  -- * comparison operators
  | Eq   Exp Exp | Neq  Exp Exp
  | SEq  Exp Exp | SNeq Exp Exp
  | Gt   Exp Exp | Gte  Exp Exp
  | Lt   Exp Exp | Lte  Exp Exp
  -- * bitwise operators
  | BAnd Exp Exp | BOr  Exp Exp
  | BXor Exp Exp | BNot Exp
  | BLs  Exp Exp | BRs0 Exp Exp
  | BRs1 Exp Exp
  -- * atomic expressions
  | Atom Atom
```

Expressions can be combined in "code", which represents expressions that can optionally have side-effects. `Code` instances are labelled.

```
data Code
  = Expr   Label Exp                -- ^ effect-free expressions
  | Assign Label Ident Code         -- ^ effectful assignments
  | Call   Label Label Ident [Code] -- ^ function calls
  | Skip   Label                    -- ^ a nop expression
```

`Code` instances can be further combined in statements, which contains all control-flow effecting expressions (aside from function calls).

```
data Stmt
  = Code   Code           -- ^ simple code expressions
  | Seq    Stmt Stmt      -- ^ sequential composition
  | If     Code Stmt Stmt -- ^ if-then-else structure
  | While  Code Stmt      -- ^ simple while loops
  | Return Code           -- ^ returns
```

Finally, statements—together with function declarations—compose a program.

```
data Decl    = Decl Ident [Ident] Label Stmt Label
data Program = Program [Decl] [Stmt]
```

Note that because of this architecture, many JavaScript structures such as e.g. nested function declarations, functions-as-values, objects, etc... are unsupported.

Analysing `Simpl`