Semester Project Idea: A Toy Structurally Typed Language

Samuel Grütter

This document is an outline of what I would like to do as a semester project. The project would consist of two phases:

- Phase I: Defining and implementing a basic structurally typed programming language which is as simple as possible.
- Phase II: Trying out different extensions, implementing them if possible, or otherwise writing a report on why it is not possible or too challenging.

1 Phase I: The basic language

1.1 General

General properties of the language:

- Similar to the TOOL language of the Compiler Construction course
- Compiler written in Scala, parts of the TOOL compiler might be reused
- One source file per program, compiled to Java class files

For the rest of this document, let `a` and `a1`, ...`aN` be identifiers, let `T` and `T1`, ...`TN` be type expressions, and let `e` and `e1`, ...`eN` be "normal" expressions.
There are two kinds of expressions: "Normal" expressions and type expressions. Fields of objects and block variables can hold the value of normal expressions or the value of a type expression. If they hold a type expressions, they only exist at compile time and are not available at run time.

1.2 Type expressions

1.2.1 Constructing type expressions

Type expressions are constructed in the following way:

- `Int`, `Bool`, `String`, `Void` and `Null` are type expressions.
- The function type `T1 -> T2` is a type expression.
- "Interface types" are constructed as follows:
  `[ a1: T1, a2: T2, ... aN: TN ]`
  The order in which the declarations inside the interface type occur is irrelevant.
- The intersection type `T1 & T2` is a type expression. If we consider types as sets of values, then `T1 & T2` means `T1 ∩ T2`.
- Fields of objects and block variables can hold the value of normal expressions, but also the value of a type expression.

Recursive types (and mutually recursive types) are not supported. In other words, each type definition can only use types defined before, and there are no forward declarations. This restriction dramatically reduces the power of the language, but on the other hand, the implementation of the compiler becomes much easier, because types cannot be infinite structures. One challenge of phase II will be to investigate if recursive types can be added to this language.

1.2.2 Simplifying type expressions

Let "type values" denote all type expressions that cannot be simplified further. All type values are have the form `Int`, `Bool`, `String`, `Void` and `Null`, `T1 -> T2`, or `[ a1: T1, a2: T2, ... aN: TN ]`. To simplify a type expression means to put it into the form of a type value if possible, or to report an error stating that the type expression is invalid. There are the following simplification rules:

- `[ c1: T1, ... cN: TN, a1: A1, ... aM: AM ] &`
  `[ c1: U1, ... cN: UN, b1: B1, ... bK: BK ]`
  simplifies to
  `[ c1: T1 & U1, ... cN: TN & UN, a1: A1, ... aM: AM, b1: B1, ... bK: BK ]`
  Here `c1, ... cN` are the common field names of the two interfaces, `a1, ... aM` are the field names that only occur in the first interface, and `b1, ... bK` are the field names that only occur in the second interface.
- If the field of an object holds a type value and occurs in a type expression, it is substituted by its type value.

If during simplification, a type expression is not a type value and none of the above rules can be applied, then the type expression is invalid. For instance, the type `String & Int` is invalid, even though it also could be evaluated to something corresponding to the empty set.
In phase II of the project, the following simplification rule might be added:

- `(A1 -> R1) & (A2 -> R2)` simplifies to `(A1 ∪ A2) -> (R1 & R2)`

However, this requires union types, and it should also be studied if this rule has any practical usefulness.

1.2.3 Comparing type expressions

The only relation on types we need is the subtyping relation. Before two type expressions are compared, they are simplified into type values. Then, as one would expect, the following subtyping rules apply:

$$\frac{}{\texttt{T <: T}} \qquad \frac{}{\texttt{Null <: T1 -> T2}} \qquad \frac{}{\texttt{Null <: [a1: T1, ...aN: TN]}}$$

$$\frac{\texttt{A2 <: A1} \qquad \texttt{R1 <: R2}}{\texttt{(A1 -> R1) <: (A2 -> R2)}}$$

$$\frac{\texttt{T1 <: U1} \qquad \dots \qquad \texttt{TN <: UN}}{\texttt{[c1: T1, ...cN: TN, b1: B1, ...bM: BM] <: [c1: U1, ...cN: UN]}}$$

1.3 Functions

There are no methods, but only anonymous functions, which are treated as expressions. All functions take one argument and return one result expression. Both of them can be the special value `void`. Functions are written as `a:T => e`. To make sure that the compiler typechecks that a function returns a desired type `R`, the function has to be assigned to a typed field, such as `f:T -> R = a:T => e`, which is a bit clumsy and might be improved in phase II.

1.4 Object construction

There are no classes, but objects are created in a way similar to JavaScript.
`(a1: T1 = e1, ... aN: TN = eN)` creates a new object with the fields `a1`, ... `aN`, whose types are `T1`, ... `TN` respectively, and which are initialized with the expressions `e1`, ... `eN`. All fields are final (final as in Java) by default. Each type may be omitted. If the type is omitted, the compiler will infer the type in a simple "bottom-up" way.
We need vals because we want to have covariant object members. We don't introduce vars for simplicity. If we still want vars, we use something like `a : var Int = Ref(5)`
To add "methods" to an object, we add a field and initialize it with an anonymous function.

1.5 Grammar

For simplicity, the precedences are not reflected in the grammar presented here. They are the same as in Java, except for function application. There are two kinds of function application: `FuncApplWithParen` and `FuncApplNoParen`. They are semantically the same, but the difference is in their precedence: `FuncApplWithParen` has higher precedence than `GetField`, which has higher precedence than `FuncApplNoParen`. Some examples to illustrate the consequences of this:

`println "Hello World"` ≡ `println("Hello World")`
`successor x.value` ≡ `successor(x.value)`
`successor(x).value` ≡ `(successor(x)).value`

Some remarks:

- Since there are no vars, there is no assignment.
- Later, the argument of a function might also be untyped.
- The interface type `[]` corresponds to Scala's type Any.
- Not all intersection types make sense, and some are not even valid, even though they pass the parser. For example, `myInt & [a: Int]` passes the parser even if `myInt` is not a type expression, but a normal expression of type Int. Only the analyzer or type checker will detect this error.
- There are no statements, but only expressions. However, the compiler might issue a warning or even an error if the value of an expression is not void and not used. The `while` construct is also an expression, even though it always returns void.
- Currently, there is no `this` keyword.
- Logical "not" is not part of the language, but can be implemented by hand by creating a function and calling it `not`.

Function application without parentheses allows us to come up with nice domain specific languages. In the corresponding example, the additional types `True` and `False` are used. They contain only one element, the value `true` or `false`, respectively. Fields which can only be `true` can be used as "markers".

```
Program ::= BlockContent

BlockContent ::= (( ValDecl | Expr ) ‘;‘)* Expr

ValDecl ::= Identifier ‘:‘ TypeExpr ‘=‘ Expr
          | Identifier ‘=‘ (Expr | TypeExpr)

TypeExpr ::= ‘Void‘ | ‘Null‘ | PrimitiveType | InterfaceType
           | FunctionType | IntersectionType | IndirectType
           | ‘var‘ TypeExpr

PrimitiveType ::= ‘Int‘ | ‘Bool‘ | ‘String‘
InterfaceType ::= ‘[‘ (Identifier ‘:‘ TypeExpr)* ‘]‘
FunctionType ::= TypeExpr ‘->‘ TypeExpr
IntersectionType ::= TypeExpr ‘&‘ TypeExpr
IndirectType ::= Identifier | IndirectType ‘.‘ Identifier

Block ::= ‘{‘ BlockContent ‘}‘
ObjConstr ::= ‘(‘ ValDecl* ‘)‘
AnonFunc ::= Identifier ‘:‘ TypeExpr ‘=>‘ Expr
           | ‘void‘ ‘=>‘ Expr
If ::= ‘if‘ Expr ‘then‘ Expr (‘else‘ Expr)?
While ::= ‘while‘ Expr ‘do‘ Expr
Println ::= ‘println‘ Expr
BinOpExpr ::=
    Expr ( ‘&&‘ | ‘||‘ | ‘==‘ | ‘<‘ | ‘+‘ | ‘-‘ | ‘*‘ | ‘/‘ ) Expr
GetField ::= Expr ‘.‘ Identifier
Literal ::= <int literal> | ‘true‘ | ‘false‘ | ‘"‘<string literal>‘"‘

ExprWithParenNoFuncAppl ::= ’(’ Expr ’)’ | ObjConstr
ExprNoParenNoFuncAppl ::= Identifier | Block | AnonFunc | If | While
                        | Println | BinOpExpr | GetField | Literal

FuncApplWithParen ::= Expr ExprWithParenNoFuncAppl
FuncApplNoParen ::= Expr ExprNoParenNoFuncAppl

Expr ::= ExprWithParenNoFuncAppl | ExprNoParenNoFuncAppl
       | FuncApplWithParen | FuncApplNoParen

Identifier      ::=     <java identifier>
```

Listing 1: Grammar of the basic phase I language

```
// ----- The Library ---------------------------------------
std = ( // something like a namespace
   IntVector = [
      size: Void -> Int,
      maxSize: Int,
      empty: Void -> Bool,
      at: Int -> var Int,
      push_back: Int -> Void
   ],
   // very limited implementation of IntVector
   newVector: Void -> IntVector = void => (
      // the following fields are private because they
      // are not in interface IntVector
      fSize : var Int = Ref(0),
      e0: var Int = Ref(0),
      e1: var Int = Ref(0),
      e2: var Int = Ref(0),
      e3: var Int = Ref(0),

      // public fields
      size: void => fSize.get(),
      maxSize: 4,
      empty: void => fSize.get() == 0,
      at: Int -> var Int = i => {
         if (i == 0) e0 else if (i == 1) e1
         else if (i == 2) e2 else if (i == 3) e3 else null
      },
      push_back: Int -> Void = el => {
         at(fSize.get()).set(el);
         fSize.set(fSize.get() + 1)
      }
   )
);
// ----- The Application ----------------------------------
println("Hello World");
println "Hello World Without Needing Parentheses";
increase = a: var Int => a.set(a.get() + 1);
v = newVector();
v.push_back(1);
v.push_back 2;
increase v.at(1);
println(v.at(1).get())
```

Listing 2: A Code Example

```
Verb = [
   isVerb: True,
   word: String
   use: Void -> Void
   printStatistics: Void -> Void
];
To = [ isTo: True ];
to : To = (isTo: true);
createVerb: wordName => (
   isVerb = true,
   word = wordName,
   usageCount: var Int = Ref(0),
   use: void => usageCount.set(usageCount.get() + 1)
   printStatistics: void =>
      println(usageCount + " times, someone " + word + " something")
)
says: Verb = createVerb "says";
gives: Verb = createVerb "gives";
Subject = Verb -> String -> To -> String -> void;
createSubject: String -> Subject =
   subjectName => aVerb => str1 => aTo => str2 => {
      aVerb.use();
      println(subjectName+" "+aVerb.word+" "+str1+" to "+str2)
   }

theProgram = createSubject("The program");
theProgram says "Hello" to "the World";

Bob = createSubject "Bob";
Bob says "Hi" to "Alice";
Bob gives "flowers" to "Alice";

gives.printStatistics();
says.printStatistics()
```

Listing 3: Domain Specific Language Example

2 Phase II: Extensions

In phase II, some extensions will be studied and added to the language if possible. Some of them might be theoretically impossible or practically too challenging and go far beyond the scope of a bachelor semester project. For those, the difficulties will be described in a report.
Some theoretical extensions:

- Recursive types
- Type abstractions / Generics. For generic functions, the type parameter would not be inferred, but had to be written explicitly.
- Covariant and contravariant subtyping of generic types
- Enumeration types: Types which are a set containing only values that are enumerated in the definition of the type.
- General union types or union over disjoint types
- Pattern matching over disjoint or general union types
- Introduce pairs, triples, ..., making sure that a $\mathtt{T} \times \mathtt{T} \times \mathtt{T}$ is not a subtype of $\mathtt{T} \times \mathtt{T}$.
- Replacing `while` and `if` by functions without making code too ugly.
- Some kind of object-level inheritance or "merging" of two objects, dealing with conflicts in field names / overriding ... in a typesafe way.
- Studying the relationship between vars (which can be read and written), readonly variables (which can change), stable values which can never change, and functions from `Void` to a value (which are similar to a readonly variable), and making access and subtyping consistant among them.

Some practical extensions, only implemented in case the language turns out to be useful:

- A toy collection framework using generics, or integerating Java Collections
- Syntax improvements such as making semicolons between statements and commas between object fields optional
- The type Real (or Double) and the type Char
- I/O functionality and access to command line arguments
- multiple source files, namespaces or packages

During the work on this project, the above lists of possible extensions would certainly grow.