Lawoco: The Spoofax submission for the Language Workbench Comparison

Introduction

This document describes the construction of the *Lawoco* domain-specific language, a simple structural entities language that is used as a comparison point in the Language Workbench Comparison of www.languageworkbenches.net. The definition of Lawoco and its documentation are available from http://strategoxt.org/Spoofax/LWC2011.

Spoofax is an Eclipse-based language workbench that focuses on textual domain-specific languages. Spoofax uses SDF for syntax definition, a modular syntax definition formalism that supports language composition. Static semantics, editor services, and code generation can be expressed using rewrite rules in the Stratego language. (More information is available at spoofax.org, syntax-definition.org, and strategoxt.org.)

For this project we use of Spoofax 0.6.1, which at the time of writing is available as a nightly build from http://strategoxt.org/Spoofax/Download#nightly. Spoofax 0.6.1 introduces new experimental features that we use in the definition of Lawoco.

Phase 0 - Basics

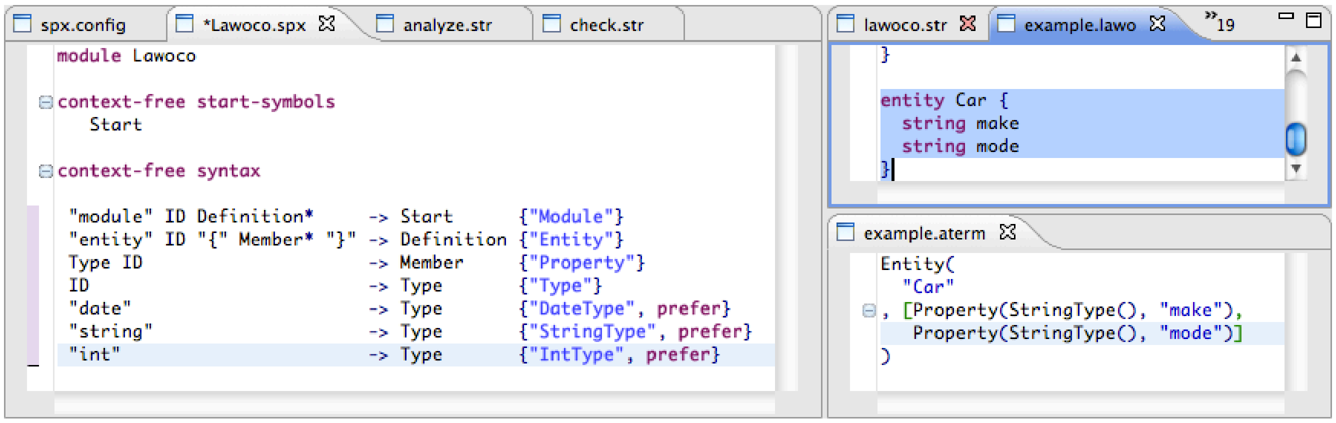
A simple structural DSL

We can define the syntax of this language using SDF as follows:

```
"module" ID Definition*      -> Start      {"Module"}
"entity" ID "{" Member* "}" -> Definition {"Entity"}
Type ID                      -> Member     {"Property"}
ID                           -> Type       {"Type"}
"date"                       -> Type       {"DateType", prefer}
"string"                     -> Type       {"StringType", prefer}
"int"                        -> Type       {"IntType", prefer}
```

Note how in SDF, productions have their pattern at the left-hand side. We also specify a label for the abstract syntax as an annotation (e.g., “Module” for modules). For composing languages and productions, SDF supports disambiguation filters such as the {prefer} annotation, which here indicates that the production should be preferred over any other matching production.

Spoofax can dynamically generate and load an editor for a given language definition. The screenshot below shows the syntax definition (left), an example Lawoco program (upper right), and the abstract syntax for the selection (lower right). Note how the abstract syntax is shown textually as first-order terms that use labels such as StringType from the syntax definition.



```
module Lawoco

context-free start-symbols
   Start

context-free syntax

  "module" ID Definition*      -> Start      {"Module"}
  "entity" ID "{" Member* "}"  -> Definition {"Entity"}
  Type ID                      -> Member     {"Property"}
  ID                           -> Type       {"Type"}
  "date"                       -> Type       {"DateType", prefer}
  "string"                     -> Type       {"StringType", prefer}
  "int"                        -> Type       {"IntType", prefer}
```

```
}

entity Car {
  string make
  string mode
}
```

```
Entity(
  "Car"
, [Property(StringType(), "make"),
   Property(StringType(), "mode")]
)
```

Code generation

Code generation can be expressed using rewrite rules on abstract syntax tree patterns as those shown in the screenshot above. Rewrite rules always take the form

r:
  before -> after

And rewrite a term pattern ‘before’ to a term pattern ‘after’. Some rules specify conditions using the ‘where’ keyword or specify actions using the ‘with’ keyword. A rule with a name ‘r’ can be called as a function using the syntax <r>. For code generation, we support string interpolation using the syntax $[...] where [ ] can be used to escape. To rewrite Lawoco to Java, we can use rules such as the following:

```
to-java:
  Module(x, d*) ->
  $[ package [x];

     [d'*]
   ]
  with
    d'* := <to-java> d*

to-java:
  Entity(x, p*) ->
  $[ class [x] {
       [p'*]
     }
   ]
  with
    p'* := <to-java> p*

to-java:
  Property(t, x) -> $[
    private [t'] [x];

    public [t'] get_[x] {
      return [x];
    }

    public void set_[x] ([t'] [x]) {
      this.[x] = [x];
    }
]
with
  t' := <to-java> t

to-java:
  StringType() -> "String"

// etc.
```

Simple constraint checks

As of version 0.6.1, Spoofax supports annotations in grammars to specify namespaces and relation. In the basic Lawoco language there are two namespaces: the *Entity* namespace and the *Property* namespace. Definition sites of a namespace can be specified in the grammar using the @= notation, while use sites can be specified using the @ notation:

```
"module" ID Definition*             -> Start {"Module"}
"entity" Entity@=ID "{" Member* "}" -> Definition {"Entity",
                                                   scope(Property)}
Type Property@=ID -> Member {"Property"}
Entity@ID         -> Type   {"Type"}
"date"            -> Type   {"DateType", prefer}
"string"          -> Type   {"StringType", prefer}
"int"             -> Type   {"IntType", prefer}
```

Together, the use sites, def sites, and the {scope(Property)} annotation specify basic constraint checks such as non-existing entity references.

We can use rewrite rules to specify additional constraints:

```
  constraint-error:
    Entity(x, _) -> (x, $[Duplicate entity definition])
    where
      all-xs := <index-lookup-all> x;
      <gt> (<length> (all-xs), 1)
```

This rule essentially rewrites every entity x to an error marker placed at the ‘x’ that says “duplicate entity definition”. It only does so under the condition that given the collection of all definitions of ‘x’, the length of the collection is greater than 1.

Modularity

Definitions can be distributed over multiple files out of the box.

Phase 1 - Advanced

1.1, 1.5 Language integration

The syntax-definition.org website has several SDF syntax definitions for languages such as C#, Java, PHP, and XML. We can embed Java expressions by importing the Java syntax into the Lawoco syntax:

```
imports Java-15
```

and by adding an additional production to that integrates Java expressions for the initialization of properties:

```
Type Property@=ID "=" Expr -> Member {"PropAssign"}
```

Because Spoofax uses a generalized parser (SGLR), conflicts such as reserved keywords in Java are avoided.

1.4 Introducing namespaces

We can introduce namespaces to Lawoco by simply introducing a new production to the syntax definition:

```
"package" Package@=ID "{" Definition* "}" -> Definition {"Package",
                                                   scope(Entity)}
```

Note how the production scopes the Entity namespace.

1.6 Multiple generators

Multiple generators can be specified using “builder” definitions:

```
builder  : "Generate Java code (for selection)" =
  generate-java (openeditor) (realtime)

builder  : "Generate XML (for selection)"       =
  generate-xml (openeditor) (realtime)

builder  : "Show abstract syntax (for selection)" =
  generate-aterm (openeditor) (realtime) (meta)
```

Each builder specifies a name, a rule to apply, and optional annotations that describe how to apply the rule. The transformations above *open an editor (openeditor)*, and are updated in *realtime (realtime)* as the source changes. The last builder is only used by *meta-programmers (meta)*.

Phase 2,3 - Non-functional and Freestyle

As additional features, Lawoco shows how to implement a basic type system, a simple expression language, and additional constraint checks.