Theme: Scrap your boilerplate

Headline
Demonstrations of SYB style of generic programming

Description
The "Scrap your boilerplate" (SYB) style of generic programming was originally conceived in a Haskell context, but similar coding styles were subsequently proposed for other programming languages. In fact, even for Haskell alone, variations on SYB style were proposed. Accordingly, the present theme features contributions that exercise SYB style across different host languages. In some cases, these contributions actually include libraries that support SYB style for the host language at hand. Certain features of the system Company are particularly relevant for the present theme. These are the features for cutting and totaling salaries which actually have their origin in the SYB literature. Thus, Feature:Total and Feature:Cut make up the baseline set of features to be covered by any member contribution of this theme. Implementations of yet other features may benefit from SYB style, too.

There are the following members in the theme:

- haskellSyb: Illustration of SYB for Haskell
- jsSyb: Illustration of SYB for JavaScript
- pythonSyb: Illustration of SYB for Python
- javaSyb: Illustration of SYB for Java

Relationships
See Theme: Haskell genericity for other generic programming styles with only Haskell-based contributions.

Metadata
- Feature:Total
- Feature:Cnt
-
System: Company

**Headline**

An imaginary HRMS system

**Description**

System:Company is an imaginary Human resource management system (HRMS) (i.e., an information system) implementations of which (‘contributions’) are documented on 101wiki. The system is supposed to model the company structure in terms of employees and possibly the hierarchical structure of departments. Employees are modeled in terms of their names, addresses, salaries, and possibly additional properties. The system is supposed to meet certain functional requirements such as totaling all salaries in the company. The system may also be subjected to non-functional requirements such as persistence or distribution. Features are not collected for the sake of an interesting HRMS system. Instead, features are designed to exercise interesting characteristics of software languages and software technologies. Most features are optional so that contributions have the freedom of choice to focus on features that are particularly interesting for a certain objective of language or technology demonstration.

There are the following features:

- **Company**: Companies, department, employees
- **Total**: Total the salaries of employees
- **Median**: Compute the median of the salaries
- **Cut**: Cut the salaries of employees in half
- **Depth**: Compute nesting depth of departments
- **COI**: Conflicts of interests for employees
- **Mentoring**: Associate mentors and mentees
- **Ranking**: Enforce salary to correlate with ranks
- **Singleton**: Constrain for a single company
- **History**: Maintain and analyze company history
- **Serialization**: De-/serialize companies
- **Persistence**: Persist companies
- **Mapping**: Map companies across technological space
- **Distribution**: Distribute companies
- **Parallelism**: Total or cut in parallel
- **Logging**: Log company changes
- **Browsing**: Browse companies interactively
- **Editing**: Edit companies interactively
- **Restructuring**: Restructure companies interactively
- **Web UI**: Operate on companies in a web browser
- **Parsing**: Parse companies in concrete syntax
- **Unparsing**: Pretty print companies

The set of all features can also be arranged in a feature model as defined by the following constraints:

- **Data requirements**
  - Feature:Company (XOR)
    - Feature:Hierarchical company
    - Feature:Flat company
  - Feature:COI
  - Feature:Mentoring?
  - Feature:Ranking?
  - Feature:Singleton?
  - Feature:History?
- **Functional requirements**
  - Feature:Total?
  - Feature:CUT?
  - Feature:Median?
  - Feature:Logging?
  - Feature:Depth?
  - Feature:Parsing?
  - Feature:Unparsing?
  - Feature:History?
- **Non-functional requirements**
  - Feature:Serialization? (XOR)
    - Feature:Open serialization
    - Feature:Closed serialization
  - Feature:Persistence?
  - Feature:Mapping?
  - Feature:Distribution?
  - Feature:Parallelism? (OR)
    - Feature:Data parallelism
    - Feature:Task parallelism
- **UI requirements**
  - Feature:Browsing?
  - Feature:Editing?
  - Feature:Undo-redo?
This specification is under construction.

We use the following informal notation here:

- $f$? means that the feature $f$ is optional.
- $f$ (OR) means that $f$ is an OR feature; any operands may be chosen, but at least one, unless $f$ is optional.
- $f$ (XOR) means that $f$ is an XOR feature; either of its operands must be selected, but not several of them.
- $f_1 \Rightarrow f_2$ means that if $f_1$ is selected then $f_2$ must be selected.
- $f$ (i.e., $f$ with strikethrough) means that the feature is only emerging or already vanishing.

**Illustration**

The following UML class diagram models the basic structure of the system.

![UML class diagram](image)

See [Theme Starter](#) for a few very simple contributions in varying languages. These are mostly implementations of the system in varying programming languages, but a UML-based model (as shown above) is also included.

**Metadata**

- Human resource management system
- Namespace:Feature
**Language:** Java

**Headline**
An **OO programming language**

**Illustration**
Let's show "Hello World" for Java.

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

**Metadata**
- **OO programming language**
- **OO programming**
- [https://java.com](https://java.com)
- **Technology_Java platform**
- **Technology_Java SE**
**Concept:** Generic programming

**Headline**

Programming with type-parametric abstractions

**Description**

The term "generic programming" is somewhat overloaded. Thus, a particular definition is established here. In a simple case, generic programming is concerned with functionality that is parameterized by a type, possibly with an associated type bound. For example, generics in Ada or Java address this form of generic programming. In a more advanced case, type parameters may also control compile-time code generation as in the case of generative programming, e.g., with template metaprogramming. Ultimately, generic programming also exploits parameterization in the shape of data types à la datatype-generic programming.

**Metadata**

- Programming paradigm
Contribution: haskellSyb

Headline
Generic programming in Haskell with SYB

Motivation
The implementation demonstrates generic programming in Haskell using "Scrap your boilerplate" (SYB) style. Generic programming is useful in so far that several operations of the 101companies features involve traversal over nested data. For instance, totaling all salaries requires the location of salary subterms at all levels in a given company term. Accordingly, operations for totaling and cutting salaries are implemented as generic functions that traverse over the company structure while type-specific cases detect employees and access their salaries. (It happens that the original publication on SYB used these generic programming samples for motivation.) The resulting code is considerably more concise, when compared to a conservative style of functional programming; see Contribution:haskellComposition. Additional operations are implemented to exercise SYB for different idioms of query or transformation and to address additional structural concerns; see the implementations of Feature:Depth, Feature:Mentoring, and Feature:Depth.

Illustration
Several of the operations on companies can be implemented in a very concise manner based on the SYB style of generic programming. For instance, the operation for totaling salaries simply extracts all floats from the given term and reduces them by addition:

```
-- Total all salaries in a company
total :: Company -> Float
total = everything (+) (extQ (const 0) id)
```

Architecture
The architecture is exactly the same as the one of Contribution:haskellComposition with one exception. There is an additional module Company.Generics which hosts stand-alone deriving clauses for instances of Data and Typeable classes, as needed for generic programming with Data.Generics.

Usage
The Haskell module Main has to be consulted with GHCi, and the main function has to be applied so that output for the test scenarios is produced. The expected output is available through the file baseline. There is a Makefile with a target test for test automation.

Metadata
- Language:Haskell
- Technology:GHC
- Technology:Cabal
- Feature:Hierarchical company
- Feature:Mentoring
- Feature:Total
- Feature:Cut
- Feature:Depth
- Feature:Ranking
- Feature:Closed serialization
- Contributor:rlaemmel
- Theme:Haskell potpourri
- Theme:Scrap your boilerplate
- Theme:Haskell genericity
- Theme:Haskell introduction
**Contribution:** javaSyb

**Headline**

SYB-style generic programming with reflection in Java

**Characteristics**

When implementing operations for totaling or cutting salaries in a regular object-oriented fashion (such as in the case of Contribution:javaComposition), it occurs that the implementations are unnecessarily concerned with the detailed object model. In contrast, some XML-based implementations (such as in the case of Contribution:javaDom) are much more concise because they can leverage extra query facilities, in fact, axes, such as the descendants axis of XPath. The SYB style of generic programming also provides such conciseness on the grounds of functional programming idioms. In particular, queries and transformations are supported by customizable traversal schemes. Conceptually, these are higher-order functions that are to be parametrized by essential, problem-specific first-order functions. The present implementation applies SYB to Java. The implementation includes a simple SYB-style library for object traversals. The library relies on reflection, in fact, introspection—as provided by Java's reflection approach.

**Illustration**

The data model is implemented with object composition. For example Companies:

```java
/**
 * A company has a name and consists of (possibly nested) departments.
 */
public class Company implements Serializable {
    private static final long serialVersionUID = -200889592677165250L;

    private String name;
    private List<Department> depts = new LinkedList<Department>();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Department> getDepts() {
        return depts;
    }
}
```

**Feature:** Closed serialization is implemented using Language:Java Technology:Object Streams:

```java
/**
 * Read (say, deserialize) a company.
 */
public static Company deserializeCompany(String filename) {
    Object o = null;
    try {
        FileInputStream fis = new FileInputStream(filename);
        ObjectInputStream in = new ObjectInputStream(fis);
        o = in.readObject();
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return (Company) o;
}

/**
 * Write (say, serialize) a company.
 */
public static boolean serializeCompany(Company c, String filename) {
    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try {
        fos = new FileOutputStream(filename);
        out = new ObjectOutputStream(fos);
        out.writeObject(c);
        out.close();
        return true;
    } catch (IOException ex) {
        ex.printStackTrace();
        return false;
    }
}
```
Feature:Total, Feature:Cut and Feature:Depth are implemented using functions. For example Total:

```java
public class Total {

    public static double total(Company c) {
        return everythingOrDefault(getSalary(), 0.0).add(0.0).apply(c);
    }

    public static Function<Company, Double> getSalary() {
        return new Function<Company, Double>() {
            public Double apply(Company x) {
                return x.getSalary();
            }
        };
    }

    public static BinaryOperator<Double> add =
        new BinaryOperator<Double>() {
            public Double apply(Double x1, Double x2) {
                return x1 + x2;
            }
        };
}
```

Test cases are implemented for all Namespace:Features.

**Relationships**

For basic OO without inheritance see Contribution:javaComposition.

For basic OO with inheritance see Contribution:javaInheritance.

For modular OO programming with static methods see Contribution:javaStatic.

For use of Java reflection see Contribution:javaReflection (data processing) and Contribution:javaSyb (SYB-style generic programming).

For design patterns see Contribution:javaTemplate (template design pattern), Contribution:javaVisitor (visitor design pattern) and Contribution:javaExorcism (excessive illustration of design patterns).

**Architecture**

The contribution follows a standardized structure:

- inputs contains input files for tests
- src/main/java contains the following packages:
  - javaf.prelude for function patterns.
  - javaf.syb for query patterns.
  - org.softlang.company.features for implementations of Functional requirements.
  - org.softlang.company.model for implementations of Feature:Company.
- src/test/java contains the following packages:
  - org.softlang.company_tests for Technology:JUnit test cases for Namespace:Features.

**Usage**

This contribution uses Technology:Gradle for building. Technology:Eclipse is supported.

See https://github.com/101companies/101simplejava/blob/master/README.md

**Metadata**

- Feature:Hierarchical company
- Feature:Closed serialization
- Feature:Total
- Feature:Cut
- Feature:Depth
- Theme:Scrap your boilerplate
- Language:Java
- Technology:JUnit
- Technology:Gradle
- Contributor:rlaemmel
**Contribution:** jsSyb

**Headline**

Generic programming in Javascript with SYB

**Motivation**

This implementation shows generic programming in Javascript using SYB style. The data model relies on (recursive) data composition alone. Generic Programming simplifies the traversal over nested data. Functions for total and cut are implemented as generic functions, they contain type specific cases for employees to extract salaries. This library is a port of SYB in JS this implementation's mapping functions are easily possible due to the simple object structure in Javascript.

**Illustration**

```javascript
function cutOne(e) {
  if (e.salary !== null) {
    return {
      salary: e.salary / 2,
      name: e.name
    };
  } else {
    throw new Error("fail");
  }
}
cut = everywhere(mkT(cutOne))
cut company = cut(company);
console.log(cut company)
```

This implementation of total takes the salary of every employee and folds them by adding.

**Relationships**

- See Contribution/haskellSyb for the haskell version
- See Contribution/pythonSyb for the python version
- See Contribution/coffeeSyb for the coffee script version

**Usage**

Use node.js to run the example, you can embedd it to an html file too.

```
$ node syb.js
12345
6172.5
```

**Metadata**

- Language:Javascript
- Feature:Hierarchical company
- Feature:Mentoring
- Feature:Total
- Feature:Cut
- Contributor:kevin-klein
- Theme:Scrap your boilerplate
Contribution: pythonSyb

Headline
Generic programming in Language:Python with SYB

Motivation
This implementation shows generic programming in Language:Python using SYB style. The data model relies on (recursive) data composition alone. Generic Programming simplifies the traversal over nested data. Functions for total and cut are implemented as generic functions, they contain type specific cases for Employees. This simplifies functional style traversing in Python a lot. This library is a port of the haskell version of SYB, but it addresses the same optimisations as those in haskell. For tree modification look at the implementation of cut and the implementation of total illustrates querys on a tree. The library depends on reflection in python to define gmapQ and gmapT.

Illustration
The data model is adapted from Contribution:py3k.

```python
@t( Employee )
def salary(e):
    return e.salary

total = everything( operator.add, 0 | mkQ | salary )
```

This implementation of total takes the salary of every employee and folds them by adding.

Relationships
- See Contribution:haskellSyb for the haskell version
- See Contribution:jsSyb for the javascript version
- See Contribution:coffeeSyb for the coffee script version

Usage
The python file "101companies.py" requires python2.5-2.7. The output appears on the console.

```
$ python 101companies.py
399747.0
199873.5
```

Metadata
- Language:Python
- Feature:Hierarchical company
- Feature:Mentoring
- Feature:Total
- Feature:Cut
- Contributor:kevin-klein
- Theme:Scrap your boilerplate
Feature: Cut

Headline
Cut the salaries of all employees in half

Description
For a given company, the salaries of all employees are to be cut in half. Let's assume that the management of the company is interested in a salary cut as a response to a financial crisis. Clearly, any real company is likely to respond to a financial crisis in a much less simplistic manner.

Motivation
The feature may be implemented as a transformation, potentially making use of a suitable transformation or data manipulation language. Conceptually, the feature corresponds to a relatively simple and regular kind of transformation, i.e., an iterator-based transformation, which iterates over a company's employees and updates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

Illustration
The feature is illustrated with a statement in Language:SQL to be applied to an instance of a straightforward relational schema for companies where we assume that all employees belong to a single company:

```sql
UPDATE employee
SET salary = salary / 2;
```

The snippet originates from Contribution:mysqlMany.

Relationships
- See Feature: Total for a query scenario instead of a transformation scenario.
- In fact, Feature: Total is likely to be helpful in a demonstration of Feature: Salary cut.
- The present feature should be applicable to any data model of companies, specifically Feature: Flat company and Feature: Hierarchical company.

Guidelines
- The name of an operation for cutting salaries thereof should involve the term "cut". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Cut.sql". Likewise, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "cut".
- A suitable demonstration of the feature's implementation should cut the salaries of a sample company. This guideline is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. Queries according to Feature: Total may be used to compare salaries before and after the cut. All such database preparation, data manipulation, and query execution should preferably be scripted. By contrast, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

Metadata
- Functional requirement
- Transformation
- Type-preserving transformation
- Iterator-based transformation
- Optional feature
- Type-preserving transformation
**Feature:** Total

**Headline**

Sum up the salaries of all employees

**Description**

The salaries of a company's employees are to be summed up. Let's assume that the management of the company is interested in the salary total as a simple indicator for the amount of money paid to the employees, be it for a press release or otherwise. Clearly, any real company faces other expenses per employee, which are not totaled in this manner.

**Motivation**

The feature may be implemented as a query, potentially making use of a suitable query language. Conceptually, the feature corresponds to a relatively simple and regular kind of query, i.e., an iterator-based query, which iterates over a company's employees and aggregates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

**Illustration**

**Totaling salaries in SQL**

Consider the following Language:SQL query which can be applied to an instance of a straightforward relational schema for companies. We assume that all employees belong to a single company; The snippet originates from Contribution:mySqlMany.

SELECT SUM(salary) FROM employee;

**Totaling salaries in Haskell**

Consider the following Language:Haskell functions which are applied to a simple representation of companies.

```haskell
-- Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = salariesEs es

-- Extract all salaries of lists of employees
salariesEs :: [[Employee]] -> [Salary]
salariesEs [] = []
salariesEs (e:es) = getSalary e : salariesEs es

-- Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary (e, s) = s
```

**Relationships**

- See Feature:Cut for a transformation scenario instead of a query scenario.
- See Feature:Depth for a more advanced query scenario.
- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical_company.

**Guidelines**

- The name of an operation for summing up salaries thereof should involve the term "total". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Total.sql". By contrast, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "total".
- A suitable demonstration of the feature's implementation should total the salaries of a sample company. This guideline is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. All such database preparation and query execution should preferably be scripted. Likewise, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

**Metadata**

- Optional feature
- Functional requirement
- Aggregation
Language: Haskell

**Headline**
The functional programming language Haskell

**Details**
101wiki hosts plenty of Haskell-based contributions. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: [Theme: Haskell data], [Theme: Haskell potpourri], and [Theme: Haskell genericity]. Haskell is also the language of choice for a course supported by 101wiki: [Course: Lambdas in Koblenz].

**Illustration**
The following expression takes the first 42 elements of the infinite list of natural numbers:

```haskell
> take 42 [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41]
```

In this example, we leverage Haskell's lazy evaluation.

**Metadata**
- Functional programming language
Language: JavaScript

**Headline**
A multi-paradigm programming language for the web et al.

**Discussion**
JavaScript counts as (imperative) **OO programming language**: it is prototype-based, though.
JavaScript provides first-class functions; hence it counts also as **functional programming language**.
The official name of JavaScript is ECMAScript.

**Metadata**
- Functional programming language
- OO programming language
- Multi-paradigm programming language
- Scripting language
Language: Python

Headline
A multi-paradigm programming language

Metadata
- Object-oriented programming language
- Interpreted programming language
- Functional programming language
- Multi-paradigm programming language
- dictionary
**Concept:** Scrap your boilerplate

**Headline**

A generic programming style

**Illustration**

"Scrap Your Boilerplate" (SYB) refers to a generic programming style where highly polymorphic function combinators are leveraged in assembling data-processing operations that recurse into compound data while being customized by type-specific cases. Consider the following implementation of Feature:Cut as part of Contribution:haskellSyb:

```haskell
cut :: Company -> Company
    cut = everywhere (extT id (/(2::Float)))
```

everywhere is a highly polymorphic function combinator which takes a polymorphic argument function to transform all immediate and non-immediate subterms of a given term of possibly any type. The actual traversal is provided by everywhere; the per-subterm functionality is to be described by the polymorphic argument of everywhere. Thus, the following type is needed:

```haskell
everywhere :: forall a. Data a => (forall b. Data b => b -> b) -> a -> a
```

In the Haskell context, the combinators for SYB turn out to be polymorphic in an interesting manner from a typing perspective: they are rank-2 polymorphic and customization requires a form of type case. The rank-2 status can be observed in the above function signature: the inner "forall" is to the left of a function arrow. The use of type case is evident from the above application of everywhere. In particular, extT models type case such that the type-specific second argument is applied when possible and otherwise the generic first argument is applied. In general terms, extT g s x translates to s x if the argument type of s equals (or generalizes) the type of x; otherwise extT g s x translates to g x.

SYB has been conceived in a Haskell setting, but the style has been applied to other programming languages as well. SYB is particularly helpful in describing all kinds of transformations and queries for compound data in a concise manner, without involving conceptually irrelevant details of the data structures involved.

**Metadata**

- [Generic programming](http://dx.doi.org/10.1145/640136.604179)
- [Boilerplate code](http://www.haskell.org/haskellwiki/Scrap_your_boilerplate)
- [Concept](http://en.wikipedia.org/wiki/Generic_programming)
Concept: SYB

Headline
Scrap your boilerplate

Metadata
- Tag: Abbreviation
**Theme:** Haskell genericity

**Headline**

Varying generic programming approaches in Haskell

**Description**

There are different classes of generic programming. The present theme is concerned with the class of generic programming that involves data type-polymorphic functions such that the functions can be applied to data of different types as, for example, in the case of the "Scrap your boilerplate" style of generic programming. The present theme is focused on different generic programming styles as they exist for Language:Haskell. Certain features of the system:Company are particularly relevant for the present theme. These are the features for cutting and totaling salaries as they illustrate the need for data transformations and queries that may need to fully traverse compound data while only some details of such data (i.e., salaries) are conceptually relevant. Thus, Feature:Total and Feature:Cut make up the baseline set of features to be covered by any member contribution of this theme.

These are the members of the theme:

- haskellSvb: "Scrap your boilerplate" style
- strafinski: Strategic programming
- haskellTree: Rose trees for representation
- tabaluga: Folds for systems of data types

**Relationships**

- See Theme:Scrap your boilerplate for a specific style of generic programming with Haskell coverage.

**Metadata**

- Feature:Total
- Feature:Cut