A framework for rapid compiler prototyping (using reflection, XML and generating Java bytecodes)

Introduction

The goal of this study is to develop a framework for rapid compiler prototyping. Following the work of [LIFE] and [VESTALE], this framework will support several programming languages "aspects" and will allow their mixing (to a certain extent, to be defined by the notion of "distance").

An aspect of a programming language can be a syntactic or semantic property or a particular feature of a programming language. For example:

Paradigms
OO
Procedural
Functional
Logical
…

Syntaxes
(Natural Language)
Smalltalk
C
Java
C++
…

Features
Aliasing
Pointers
Blocks
Inner classes
Partial evaluation
Memo-functions
Invariant
(No) Types
Variables declarations
…

The distance between aspects of programming languages will clearly define if two aspects are compatible (possibly using mathematical expressions). Questions such as:

- « Is it possible to combine functional and object-oriented programming? »
- « Is it possible to combine logical and procedural programming? »
- or « Is it possible to combine C-like syntax and Eiffel-like syntax »

Will then find rigorous answers. Different approaches to calculate this distance (use of arbitrary weights (see next section), writing of translators and comparison of their length/speed/efficiency…) will be studied.

In parallel, a library of abstract elements will be built. Those elements will combine into a framework to define new compilers.

Finally, the distance between languages will be used in association with the library of abstract elements to create compilers understanding languages with mixed aspects. The use of XML to define the elements to be used may be interesting to study, in relation with formalisms and systems such as those describe in [MINOTAUR].

Distance

One way to calculate the distance between languages is the use of arbitrary weights. **Table 1** shows three categories of aspects with their associated weights. A weight is a set on N (where N is the set of all integers): $[\alpha, \beta]$, with $\alpha \in N$, $\beta \in N$ and $\alpha \leq \beta$. Such a set defines the importance of its related aspect in the space of all the aspects. For instance, the first category presents four different paradigms. The paradigms of Object Oriented (OO) programming and Procedural programming have respectively [0, 2] and [0, 1] as weights. Those two sets overlap since the Procedural paradigm is included into the OO paradigm, i.e. one can write programs in a procedural manner using an OO programming language. The Functional paradigms (respectively Logical paradigm) has its weight reduced to a point different from the other weights since it is a radically different paradigms, i.e. it has no intrinsic common part with other paradigms. The same rules apply to the other categories. The values chosen have no meaning by themselves; they only have a signification relatively to the other values.

Paradigms	Weights
OO	[0, 2]
Procedural	[0, 1]
Functional	[3, 3]
Logical	[4, 4]

Syntaxes	Weights
(Natural Language)	[0, 5]
Smalltalk	[1, 1]
C	[2, 5]
Java	[3, 5]
C++	[4, 5]

Features	Weights
Aliasing	[1, 1]
Pointers	[2, 2]
Blocks	[3, 3]
Inner classes	[3, 4]
Partial evaluation	[5, 5]
(No) Types	[6, 6]
Invariant	[7, 7]
Memo-functions	[8, 8]
Variables declarations	[9, 9]

Table 1: Three categories of aspects and their weights

Once the weights chosen, a multidimensional graph (or several graphs) can be drawn given two specific languages. The graphs representing the differences between Smalltalk and Java are shown **Figure 1**.

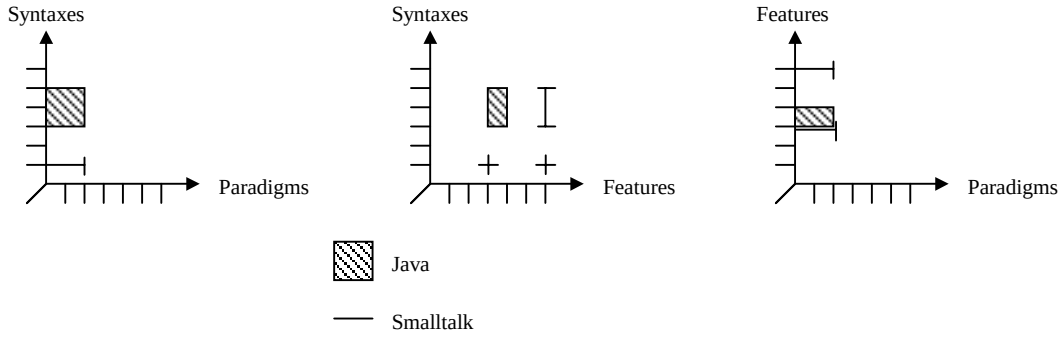

Figure 1: Graphs of aspects for Java and Smalltalk

The first graph shows how the paradigms and syntaxes of Smalltalk and Java are related. The square corresponds to the intersection of the two sets of weights of Java, while the line is the same intersection for Smalltalk. The graph explicitly shows the relation between Java and Smalltalk. Those languages are both OO programming languages (the line is right below the square and has the same position on the Paradigms axis) but they have very different syntaxes (the line is outside the square on the Syntaxes axis).
Such an analysis can be performed on the second and third graphs. The third graph also presents a case of contact between the aspects of Java and Smalltalk. The line $[0, 2] \times [3, 3]$ represents Smalltalk's OO aspect and blocks feature, while the square $[0, 2] \times [3, 4]$ is Java's OO aspect and inner classes feature. Since blocks and inner classes derive from a same concept, it makes sense for the line and the square to be connected.
From those graphs, it can be deduced that Java and Smalltalk are two languages quite different from one another, thus it should be relatively easy to mix them within a same compilation unit.

However, there are important points to be considered:

- Separate aspects of languages may not be independent, i.e. the semantic behaviour may not be independent from the syntax: aspects may no be "orthogonal" with each other. This is the reason why each category of aspects is considered in relation with the other categories (Paradigms with Syntaxes, Syntaxes with Features and Features with Paradigms).
- The semantic of those graphs must be carefully defined, i.e. if a graph shows no overlapping, it possibly means that those aspects can be combined, but to what extent? And what does it mean when aspects overlap?
- The weights and aspects presented here must be re-written to obtain the most possibly exhaustive and coherent list. This study may show some limits to the systems: what if an aspect needs to be defined as an intersection of weights?

XML

The eXtensible Markup Language (XML) is a useful way to describe structured documents. XML can also be used to encode knowledge or data. Despite its lack of semantic (some work is being done on that matter), it's a powerful way to describe hierarchy of objects and relations between them. Thus, it will be used to describe new compiler. For each compiler to be generated, a Compiler Description File (CDF) (in XML) and a grammar (Attribute Grammar) are written. The compiler generator (which uses the library of component mentioned above) parses them, and generates a new compiler (or, at least, its core framework) that supports the given grammar and CDF.

I don't know what this CDF will look like yet, but as far as my researches went, I think that it will describe some particular properties of the to-be-generated compiler and will include some specific options. The grammar and the CDF could maybe be combined into one file using the Minotaur [MINOTAUR], Centaur or FNC-2 systems (systems used to specify syntax and semantics of programming languages and generate efficient semantic tools from these specifications).

Bibliography

[LIFE] H. Ait-Kaci, Bruno Dumant, Richard Meyer, Adreas Podelski, Peter Van Roy; The Wild LIFE Handbook (prepublication edition); http://www.isg.sfu.ca/life/; March 1994
[VESTALE] http://www.enseeiht.fr/Recherche/Info/Logiciel/vestale/vestale.html
[MINOTAUR] Isabelle Attali and Didier Parigot; Integrating Natural Semantics and Attribute Grammars: the Minotaur System; INRIA, Rapport de Recherche no 2339; September 1994

Conception par objets d'un système pour combiner raisonnement formel et satisfaction de contraintes; Anne Liret, Pierre Roy and François Pachet; LIP6, Paris, France; February 1998
Loïc Correnson, Etienne Duris, Didier Parigot and Gilles Roussel; Schéma générique de développement par composition; Poitiers – Futuroscope; 1998.

Thanks

I would like to thank warmly Philippe Mulet for his advice on this draft and his support.