Sydney Area Programming Languages INterest Group Towards Improved Abstractions for Programming Language Processor Specification

Shirley Goldrei, Macquarie University

16th October 2006

The expression of programming language semantics at a high-level which enables efficient automatic generation of compilers remains an open research problem. A number of systems have been developed and active research continues, however these systems are very rarely used in practice in industry. Most programmers feel uncomfortable or unwilling to learn the existing systems and fall back on hand coding translation algorithms in an ad-hoc manner using whichever general purpose programming languages they use for all other components of the systems they develop.

The existing systems generally divide into one of two categories of semantic expression. Attribute Grammars on the one hand and term or tree rewriting systems on the other.

Attribute Grammars have particular strengths in describing analysis problems, since the language designer focuses on establishing dependencies and relationships between attributes of nodes in the syntax tree and the tools automatically compute tree traversals and attribute evaluation orderings. However Attribute Grammar systems are relatively weak in describing transformations which are based on these analyses. Transformations in an Attribute Grammar system require the language designer to explicitly build an output tree from scratch rather than describing local sub-tree transformations.

On the other hand transformation systems were precisely designed to describe local sub-tree transformations, hence they do this exceedingly well. However their weakness lies in that these systems either implement a fixed set of orderings in which transformations are applied or else the order needs to be explicitly described by the language designer.

The strengths of one type of system are precisely the weakness of the other. It therefore seems natural to attempt to combine the two approaches, however only recently has there been any attempt made to do so.

In this research I plan to consider different ways of combining these two approaches to make the whole more accessible and useful to programming language developers, especially for domain specific languages.

What often differentiates different forms of language processor is how different the input language is from the output language. Traditional compilers transform a relatively high-level language to a very low-level such as assembly language. Other language processors may translate between dialects of the same language or from one high-level source language to another high-level source language. The amount of analysis and computation required is understandably a function of the semantic as well as syntactic difference between the source and target languages - the *semantic and syntactic gap.*

There is no accepted method of quantifying the size of a semantic or syntactic gap, yet shown two examples of such a gap we may arguably be able order them from smaller to larger.

As a starting point for this research project I have devised two tiny domain specific programming languages. The design criteria for these two languages was as follows:

1. be applicable to the same domain
2. be small enough that their grammar and semantics can be easily described in limited space
3. be syntactically and semantically different enough from each other that to transform a program would require:
    (a) non-trivial rearrangement of program syntax structure
    (b) some analysis or computation to preserve semantics

These languages form the basis for experiments to explore and compare techniques for programming language translation.

In this talk I will describe the experiences gained to-date using existing techniques for program translation and discuss planned future work.