An Algebraic Language Processing Environment

Teodor Rus, Tom Halverson, Eric Van Wyk, and Robert Kooima

Department of Computer Science, The University of Iowa, Iowa City, IA 52242

1 Introduction

Problem domains evolve, so it seems natural that the languages of problem solving need to evolve as well. The theme of our research is that the task of language design and implementation should be made a realistic endeavor for a larger group of computer users thus supporting the evolution of the language with the problem domain. This can be accomplished by creating a language processing environment in which methodologies and tools are provided that simplify and ultimately automate many of the language processing tasks. This reduces the effort necessary on the part of the language designer to develop a language and create language processing tools. Since the language itself is the key to problem solving, this is a prerequisite to further progress.

A language is a communication mechanism which provides a framework in which one can create statements which have meaning. In the realm of computer systems, a language is a notation used to express computations. The notation is referred to as the syntax, while the computation is the semantics. Computer language manipulation implies a specification aspect, a processing aspect, and a usage aspect: specification allows communicators to construct valid phrases that may be used to denote computing abstractions; processing concerns algorithms that recognize phrase validity, discover the computing abstractions denoted by a phrase, and perform meaning-preserving mappings of a phrase within a language and between languages; usage is the process of problem solving using computer languages. A *language processing environment* is a set of integrated tools that supports these aspects of computer language manipulation. Though these aspects of language manipulation are intimately related, historically they have been developed into software systems by different people using different approaches. Our project advocates a unifying framework for dealing with languages in which all aspects of language manipulation are formally specified. Specifications should formally define syntax and semantics by the same specification rules and should be used to automatically generate language processing tools, thus removing most of the usual programming burden from the developer. This simplifies the problem solving process and assures that correct specifications produce correct implementations. In addition, the language developer, as well as the language user, should be able to build processing systems incrementally and interactively.

Our vision of a language processing environment has resulted in the continuous development and evolution of the TICS (Technology for Implementing Computer Software) project. This project demonstrates many unique specification and processing components as well as new methods of integration which are

based on the algebraic concepts of compositionality, incrementality, and modularity. Publications, tutorials, and demonstration materials may be found at http://www.cs.uiowa.edu/~rus. This methodology shows great promise: it has been applied to problems in a wide range of areas including syntactic[1] and semantic[2] analysis, language to language translation[3, 4], and the integration of model checking algorithms into the compiler as tools used for code optimization and parallelization[5, 6].

2 Specification

A language specification should formally describe the language syntax, semantics, and the relationship between them. In our framework, the signature of an operation in the source language algebra is described by a specification rule written in the well understood BNF notation $r : A_0 = t_0 A_1 t_1 A_2 t_2 \dots A_n t_n$, where A_i, $0 \leq i \leq n$, are parameters called nonterminal symbols and t_i, $0 \leq i \leq n$, are fixed strings called terminal symbols. A collection of computation laws or interpretations of that signature which embeds the source algebra operations into various target algebras is attached to each specification rule. Thus, the syntax and the semantics of a language construct are specified by the same rule. While the BNF notation is common, our interpretation is not. We interpret each parameter A_i, both as a semantic domain, in which A_i is a set of computation objects denoted $[\![A_i]\!]$, and as a syntactic domain, in which A_i is the set of language phrases denoted $[A_i]$. Furthermore, each $w \in [A_i]$ represents a computation object $c_w \in [\![A_i]\!]$. Accordingly, each BNF rule is interpreted both as the signature of a syntactic operation $t_0 t_1 \dots t_n : [A_1] \times [A_2] \times \dots \times [A_n] \rightarrow [A_0]$ whose computation law is $t_0 t_1 \dots t_n(w_1, w_2, \dots w_n) = t_0 w_1 t_1 w_2 \dots t_{n-1} w_n t_n \in [A_0]$, for each $w_i \in [A_i]$, $1 \leq i \leq n$, and as the signature of a semantic operation $t_0 t_1 \dots t_n : [\![A_1]\!] \times [\![A_2]\!] \times \dots \times [\![A_n]\!] \rightarrow [\![A_0]\!]$ whose computation law depends on the language processing purpose.

For each processing purpose, the computation specified by the semantic operation $t_0 t_1 \dots t_n : [\![A_1]\!] \times [\![A_2]\!] \times \dots \times [\![A_n]\!] \rightarrow [\![A_0]\!]$ is represented by a semantic macro associated with the BNF rule. Algebraically this macro is a derived operation embedding the language objects specified by the BNF rule into the language that supports the operations used in the macro-operation. In programming jargon, semantic macros are similar in form to conventional macros[7]. The difference results from the purpose of these macros. Conventional macros are used to extend the source language with user defined constructs; our macros are used to implement the source language by expressing the semantics of its valid constructs using parameterized constructs of the target language. The parameters of a semantic macro are valid target language constructs that implement the components of the source language constructs. Thus, the target image of a construct is built by the macro processor in a compositional manner as the source language construct is recognized. Consequently, semantic macros are automatic programming mechanisms that generate target programs from target components in a manner similar to that in which source language programs are generated as alge-

braic expressions by operations in the source language algebra. As an example, language translation operations are defined as computation laws performed by the operation $t_0 t_1 \ldots t_n : [\![A_1]\!] \times [\![A_2]\!] \times \ldots \times [\![A_n]\!] \rightarrow [\![A_0]\!]$ where objects $c_i \in [\![A_i]\!]$ are target images of the constructs $w_i \in [A_i]$, $1 \leq i \leq n$, and the result is the target image of the construct $w \in [A_0]$ such that $w = t_0 w_1 \ldots t_{n-1} w_n t_n$. Hence, this macro is a derived operation in the algebra of the target language and it embeds the language of $[A_0]$ into the target language by constructing the target image of w in terms of the target images of its components.

The macro-processor of semantic macros does not interfere with the parser of the source language because it is used for target image generation. Semantic macro-processors perform actions similar to those performed by the assembly language macro-processors since they produce correct pieces of target code from correct target code components. In addition, a semantic macro processor may check semantic properties of source and target language constructs, such as type, to determine exactly what should be produced as the result of the macro expansion. In other words, a semantic macro is more than just a substitution of the parameters. This has proven quite useful and powerful in a wide range of applications, as seen above.

Thus, the specification methodology we advocate is to combine a collection of compositional specification fragments into a specification rule. This takes the form of a syntactic rule and a group of zero or more macros, each for a particular language processing purpose such as mapping to a target language, defining the semantics of the construct in terms of the semantic of construct components, or constructing a graph representation of the computation.

3 Processing

Language processing tasks encompass such activities as recognizing the validity of language phrases, discovering the meaning of a message by creating an understandable representation of the intent of the communication, and mapping phrases or representations into other forms. As alluded to in the introduction, this takes the shape of extracting information from the specification rules to guide the activities of various algorithms. For example, the BNF rules are preprocessed to collect context information[1] which is used to guide the behavior of the parser to determine if its input is a construct of the language. As an aside, a jumping pattern matching parser[4] is used for construct recognition. It is a bottom up recognizer which may make reductions (rewriting) anywhere in the input as long as the validity of the input is maintained. Also, each macro in the specification is processed by the TICS system itself to create an internal, usable form. This processing produces the computation law that guides the semantic interpretation of the signature. When the macro is expanded, it will perform the intended language processing action such as translation or optimization.

This mechanism of developing language processing tools raises the issue of compositionally building a valid language phrase from valid subphrases. Since in our case, macros are used to map valid source language constructs to valid target

language constructs, this issue becomes one of how to combine target language fragments while maintaining the validity of the result. This mechanism of target code generation ensures correct bootstrapping of language implementation where individual language processing tools are generated automatically from the specification rules.

4 Integration

The objective is to use a combination of the language processing tools to operate with language elements to achieve a given goal such as syntax analysis, semantic analysis, dataflow analysis, and program translation.

The BNF portion of a specification rule describes the syntax of source language constructs. It is used to generate the parser which we call a language recognizer because it can incrementally recognize the validity of any language construct in its input. When the parser recognizes the validity of a construct in the input using a rule r, it will perform a rewriting. Also, it communicates this information to other tools which deal with the macros. The integrating environment is a data structure we have tentatively termed an Abstract Parse Tree (APT) which is constructed by the recognizer. Each node in the APT is labeled by the rule r that was used to construct it, and its children are the APT nodes for the parameters on the right hand side of r. Each macro is provided with a hook on the APT so that it can attach the information generated during macro-expansion that is carried along for access during further processing activity. The APT node is given to a macro integration function to properly activate the macro processing and the recognizer may continue in parallel with the macro-processing since it does not partake in this activity. The integration function describes how to use various macros to achieve the goal.

The APT represents the syntactic structure of the input, while each macro creates a graph representation of the computation whose nodes are attached to the APT and whose edges are semantic relationships between the nodes as constructed according to the goal of the macro. For example, the semantic analysis macros generate a structure which hangs on the APT and represents the semantics of the construct recognized by the node. The interpretation of this structure is the transition system embodying the computation of the construct. The optimization macros generate another data structure superposed on the APT which represents a model of the computation encapsulated in the construct recognized by the APT node; the nodes of this structure are processes and the edges are dependences among these processes. The code generation macros superpose over the APT yet another structure whose nodes are the target code representing the computations at that node. In this way, all of the macros operate in a common environment, yet no restrictions are placed on what they do.

5 Demonstration

Our vision of a language processing environment becomes a unifying framework for dealing with languages. The TICS system provides tools and methodologies to simplify the task of developing a language and implementing language processing tools. A system demonstration, to be sketched below, will provide an insightful look at the advantages inherent in the use of our approach.

First the syntax of a simple block structure language will be developed. From this, we can construct a recognizer which decides the validity of a language phrase. Further, we associate a semantic macro with each rule developed so far to demonstrate the semantic analysis by mapping each construct into a representative transition system. This specification is used to generate a tool which performs the semantic analysis task. The two tools are then integrated and the result of this integrated tool will be a transition system superposed on the APT. Next, we associate another macro with each rule to describe a graph representation of the programs written in this language. We may now integrate the resulting tool with the recognizer and semantic analyzer obtaining a tool which creates the model of the program over which we may apply a model checking algorithm to discover program properties[5, 6]. Finally, yet another semantic macro may be attached to each rule to describe how to construct a target language image of the construct defined by the rule in terms of the target images of the construct's components. Automatically, we could create a tool to demonstrate all of the major language processing tasks: validation, understanding, and mapping.

The value of this experiment resides in the fact that the results can be reused in real-life language processing. We will show how is this carried out on subsets of such languages as C, Java, and Fortran.

References

1. T. Rus and T. Halverson. Algebraic tools for language processing. *Computer Languages*, 20(4):213–238, 1994.
2. T. Rus. Algebraic processing of programming languages. In A. Nijholt, G. Scollo, and R. Steetskamp, editors, *Twente Workshop on Language Technology*, pages 1–42, University of Twente, Enschede, The Netherlands, 1995.
3. T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.
4. J.L. Knaack. *An Algebraic Approach to Language Translation*. PhD thesis, The University of Iowa, Department of Computer Science, Iowa City, IA 52242, December 1994.
5. T. Rus and E. Van Wyk. Integrating temporal logics and model checking algorithms. In *Fourth AMAST Workshop on Real-Time Systems, Proceedings, Lecture Notes in Computer Science, 1231*. Springer-Verlag, May 21 1997.
6. T. Rus and E. Van Wyk. A formal approach to parallelizing compilers. In *SIAM Conference on Parallel Processing for Scientific Computation, Proceedings*, March 14 1997.
7. D. Weise and R. Crew. Programable syntax macros. *ACM SIGPLAN Notices*, 28(6), 1993.

This article was processed using the LaTeX macro package with LLNCS style