# A Semantic Approach to the Solution of the Legacy Code Problem

Alex de V. Garcia
e-mail:alex@inf.puc-rio.br

Edward Hermann Haeusler
e-mail:hermann@inf.puc-rio.br

Armando M. Haeberer
e-mail:armando@inf.puc-rio.br

PUC-RioInfMCC21/97 June, 1997

**Abstract.** This work proposes a semantic approach to the legacy code problem and presents preliminary results based on a working prototype. A functional specification of the source code is obtained by means of the semantic specification of the language in which it is written. The specification is then translated into an object oriented language. Both tasks are achieved by means of transformation tools.
**Keywords:** transformation systems, denotational semantics, compiler generation, software reengineering, software reuse.

**Resumo.** O presente trabalho propõe uma abordagem semântica para a solução do problema do *legacy code* e apresenta resultados preliminares baseados em um protótipo. Uma espicificação funcional do código fonte é obtida por meio da descrição semântica da linguagem na qual está escrito. Esta espicificação é então traduzida para uma linguagem orientada a objetos. Ambas as tarefas são realizadas pelo uso de ferramentas de transformação.
**Palavras-Chave:** sistemas transformacionais, semântica denotacional, geração de compiladores, reengenharia de *software*, reuso de *software*.

# 1 Introduction

The so-called "legacy code problem" is the problem of rescuing those large systems that, after prolonged use, have been adjusted and tuned to a given specification and to the requirements of the environment where they are immersed. Such systems are *rarae avis* in the unreliable world of software products. Nevertheless, as a system becomes mature after some cycles of tuning and testing, it is likely that the software/hardware platform over which it has been implemented becomes obsolete, and its maintenance cost grows in a way that makes its use not compensating.

Whereas one can postpone the migration of the system to a modern base, the old platform must be abandoned eventually. Even if there is an up-to-date documentation of the system requirements (which is seldom the case), a new implementation of the system would be strenuous and expensive, and it is unlikely to maintain the original reliability.

In this paper, we shall describe the implementation of a semantic translating system — STS — that translates any language into a fixed object oriented language. The STS approach is based on a semantic specification of the language in which the legacy code is written. The denotation function is applied to the source code in order to produce a semantic specification of the program. This specification is then translated into the object oriented language.

The user of STS does not specify a translation by writing imperative programs or transformations, but only by providing the denotational specification of the source language. It is shown that whereas in a usual translating system it is necessary to prove its correctness for each source language, in STS a single correctness prove assures that the meaning of the input program is preserved for any source language.

The work described herein resumes the research started in 1993 at the Laboratory of Formal Methods of the Department of Informatics at Pontificia Universidade Católica do Rio de Janeiro. Hence most of the ideas in this work can be found in [HHC93], the majority of the new ideas came up as unfoldings of the implementation feedback.

## 1.1 Solutions to the Legacy Code Problem

Some common solutions to the legacy code problem are summarized here:

– Specifying and constructing a new system. This approach offers an opportunity to redesign the system based on the previous experience. Hence, this solution may lead to a more homogeneous new project, but it is the most expensive choice, for the former work is discarded.
– Reengineering and rebuilding the system. This method starts from the old system and its documentation as a first specification of the new one. It is closely related to the next solution, but here we distinguish the case *without* source translation from the one *with* source translation. The requisite of up-to-date documentation is hardly accomplished.
– Translating source code to the new platform. The source code is an excellent documentation of the system's functionality, as it describes exactly what the system does and how it works. Therefore, one may consider automatically translating the system's source code to the new platform. This method starts from a trustworthy and complete specification (the source code), therefore having enough information to eventually produce a well-suited new system. In fact, many reports of implementations of such translations have been made as in [GS93], [LSA95], [Oli96], among others.

Of course, one can combine two or all of the solutions mentioned above. It is possible that part of the system has quality to be automatically translated, another part has to be reengineered, and the remaining part has to be specified and built from the beginning. Section 2 focuses on the third solution, i.e., translation, from which we will develop our approach. From now on, we shall use **source language** ( $L_s$ ) to mean the language on which the legacy system is currently written, and **target language** ( $L_t$ ) to mean the language into which the translation is desired. There are several methods to translate a system from $L_s$ to $L_t$. They will be analyzed in detail in section 2.

### 1.2 Paper Organization

This paper is organised as follows:

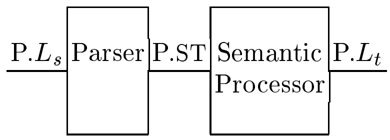
- Section 2 examines how automatic source translation is commonly implemented.
- Section 3 concisely introduces the TXL transformation tool.
- Section 4 presents the functional language used to specify denotational semantics within this paper.
- Section 5 describes the implementation of STS system in depth.
- Section 6 draws considerations on the target code correctness.
- Section 7 presents results and conclusions.

## 2 Automatic Translating

In this section this particular solution of the legacy code problem shall be described in greater detail.

### 2.1 Compiling

Compiling is the most usual way of translating from one language into another. The compilation process can be viewed as:




- P.$L_s$. Program written in the source language.
- P.ST. Syntax tree of the same program.
- P.$L_t$. Equivalent program written in the target language.

In this method the syntatic specification (grammar) of the source language is used to create the parser, whereas the semantic processor, which is the implementation of the meaning of P.$L_s$, is built manually from a semantic specification (usually non-formal) of the language. In the case of legacy system translation, there are special conditions. First, $L_t$ is usually a high level language, therefore it is easier to bestow meaning to a construct of the source language. Besides, as the legacy system is a tested program, one may give up static verifications such as type checking and scope checking. Based on these conditions, our semantic processor can be reduced to a tree-manipulator, i.e., a program that changes patterns of the program's syntax tree in P.$L_s$ to patterns of a syntax tree in P.$L_t$.

### 2.2 Transforming

Many compiler constructing tools which offer tree manipulation and pattern matching facilities have been developed in the last decades. They suggest a method of programming, the transformational paradigms (for a definition of programming paradigm, see [CH96]). A brief outline of these tools follows:

**Refine** A commercial system produced by Reasoning Systems. [REF92]
**Kimwitu** Allows the construction of programs that use trees and terms as their main data structure.
**Optran** A compiler generator with a powerful support to attribute grammars, pattern matching and transformations.

**Rigal** A compiler construction language. Its main data structures are atoms, lists and trees. Control structures are based on pattern matching.

**Popart** A system with a parser definition language and rewriting rules.

**Smart** An ANSI C extension to graph and tree algorithms.

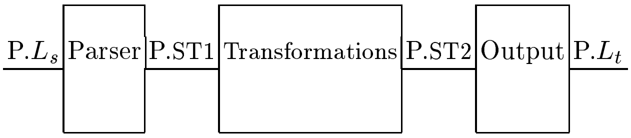
**TM** A template and structure pre-processor that generates code to a specified language.

**Tampr** A system written in Lisp. Most of the applications reported have been in porting Lisp to Fortran. [Boy89]

**Draco-PUC** It is a powerful tool developed in the Department of Informatics at Pontificia Universidade Catolica do Rio de Janeiro. It aims at implementing the Draco paradigm for software construction. [LSF94]

**Txl** This transformational programming language has a simple description language and very good documentation [CC93]. It has been chosen to implement STS.

The figure below ilustrates the use of transformations to translate code from $L_s$ to $L_t$:




- P.$L_s$. Program written in the source language.
- P.ST1. Syntax tree of the same program in $L_s$.
- P.ST1. Syntax tree of the program in $L_t$.
- P.$L_t$. Program written in the target language.

The parser can be one used in a compiler. The transformations are a program written as rules which describe associations of patterns in the syntax tree of the program in $L_s$ to patterns in the other syntax tree.

In this solution, the transformations provide the meaning of constructions of $L_s$ by translating them to $L_t$. In this case, as in the previous one (compiling), the semantics of the target program is given in an *ad hoc* basis, i.e., the meaning of a construction is coded in a programming language, and its accuracy depends upon the programmer's skill, as well as on his/her understanding of the source language informal specification.

## 3 The TXL Transformation Tool

In this section the TXL transformation tool is presented. Txl was chosen to implement STS because of its elegant language and reliable documentation [CC93]. The execution of a TXL program has three phases:

**The Parsing Phase** Parses the input, producing a parse tree. TXL can parse any context-free language. As known from compiler theory, there is no parsing algorithm efficient (with linear complexity) to all context-free languages. TXL parse is efficient, provided that the grammar has no left recursion.

**The Transformation Phase** Acts in the parse tree, transforming it according to the transformation functions and rules.

**The Unparsing Phase** Simply transverses the transformed parse tree, generating the output of the program. The output can be formatted by means of special symbols in the grammar.

A TXL program must provide a syntax definition of the input language (so that the input can be parsed), and, obviously, a specification of the set of transformations to be performed on the parse tree of the input. In the following subsections, the TXL programming language will be described in detail.

### 3.1 The Grammar Definition Part

The grammar is specified in a language similar to Extended Backus-Nauer Form (EBNF). The following example shows the TXL code for an Expression grammar and the corresponding BNF productions. "Program" denotes the initial symbol, and "number" is a special terminal that matches numeric constants.

| TXL | BNF |
|---|---|
| define program<br>  [E]<br>end define | program ::= E |
| define E<br>  [T] + [E] \|<br>  [T]<br>end define | E ::= T + E \| T |
| define T<br>  [F] * [T] \|<br>  [F]<br>end define | T ::= F * T \| F |
| define F<br>  [number] \|<br>  ( [E] )<br>end define | F ::= number \| ( E ) |

### 3.2 The Transformations

In order to write a calculator in TXL which evaluates the expressions described in the grammar above the appropriated transformation rules would be:

| Transform | Into |
|---|---|
| [N1] + [N2] | [N1 + N2] |
| [N1] * [N2] | [N1 * N2] |
| (N1) | N1 |

The simplest transformation in TXL has the form:

> **rule** *rulename*
>   **replace** [*type*]
>   *pattern*
> **by**
>   *replacement*
> **end rule**

This rule works by first trying to match *pattern* in the rule's scope, if it succeeds, it changes the input *pattern* by *replacement*. This operation is repeated until no further matches can be found in the scope. Both *pattern* and *replacement* must be of type *type*, i.e., it must be possible to parse them as *type*, according to the grammar. Now the expression evaluation transformations can be written in TXL:

```
rule rAdd
  replace [E]
    N1 [number] + N2 [number]
  by
    N1 [+ N2]
  end rule

rule rMul
  replace [T]
    N1 [number] * N2 [number]
  by
    N1 [* N2]
end rule

rule rPar
  replace [F]
    ( N [number] )
  by
    N
end rule
```

The brackets [ ] denote the type of the variable when written in the *pattern*. Otherwise, they denote the name of a rule to be applied to the preceding variable. (This variable will be the scope of the rule when it is executed.)

The starting point of the execution of a TXL program is the special rule "mainRule" that is triggered when execution starts and whose scope is the whole input of the program. In this case, "mainRule" invokes the other functions until no further transformations can be performed.

```
rule mainRule
replace [E]
    E1 [E]
  construct E2 [E]
    E1 [rAdd] [rMul] [rPar]
  where not
    E2 [= E1]
  by
    E2
end rule
```

### 3.3 Translating Programs Using TXL

The TXL tool has been used to various tasks. We are specially concerned with using TXL to translate programs between different languages. It is shown herein three methods for translating languages using TXL. For sake of simplicity it is shown how to translate expressions into postfixed expressions.

**Grammar Merging** This is perhaps the most direct way to translate form one language to another with TXL. In other to use this method, grammars of the source and target languages must be similar (as in the case of expressions and postfixed expressions). The merged grammar is:

| TXL | BNF |
|---|---|
| define program<br>[E]<br>end define | program ::= E |
| define E<br>[T] + [E] \|<br>[T] \|<br>[T] [E] +<br>end define | E ::= T + E \| T<br>\| T E + |
| define T<br>[F] * [T] \|<br>[F] \|<br>[F] [T] *<br>end define | T ::= F * T \| F<br>\| F T * |
| define F<br>[id] \|<br>( [E] )<br>end define | F ::= [id] \| ( E ) |

This grammar does not has a parenthesesless construction of F, as F ::= E. This option would make the grammar have left recursion and make the parse not efficient. In this case, it is better to remove parentheses in a separate step. The transformations are:

```
rule rAdd % Replaces a sum by its postfix version
    replace [E]
        T1 [T] + E1 [E]
    by
        T1 E1 +
end rule

rule rMul % Replaces a product by its postfix version
    replace [T]
        F1 [F] * T1 [T]
    by
        F1 T1 *
end rule

rule mainRule % Iterates the previous rules
    replace [E]
        E1 [E]
    construct E2 [E]
        E1 [rAdd] [rMul]
    where not
        E2 [= E1]
    by
        E2
end rule
```

**Using Flat Grammar** Another way to translate between to languages in TXL is to disregard the grammar of the second language (we supose that the transformation rules write correct code) and see a string in the target language as simply a list of tokens. In this case the grammar of the source language is combined with the flat grammar to the target language, as in the following example:

```
define program
        [E] |
        : [repeat any_token]
end define

define E
        [T] + [E] |
        [T]
end define

define T
        [F] * [T] |
        [F]
end define

define F
        [id] |
        ( [E] )
end define

define any_token
        + | * | id | ( | )
end define
```

The colon is used only to avoid ambiguity. The transformations are also simple, functions are used instead of rules. Functions work just like rules, but require that the full scope matches the pattern and perform the replacement only once. In this case functions are used to transverse the parse tree, while generating the target code. The "p" function is an overloaded splice fucntion.

```
function mainRule
        replace [program]
                E1 [E]
        construct AUX [repeat any_token]
        construct R1 [repeat any_token]
                AUX [rE1 E1] [rE2 E1]
        by
                : R1
end function

function rE1 E0 [E]
        replace [repeat any_token]
                INI [repeat any_token]
        deconstruct E0
                T1 [T] + E1 [E]
        by
                INI [rT1 T1] [rT2 T1] [rE1 E1] [rE2 E1] [p +]
end function

function rE2 E1 [E]
        replace [repeat any_token]
                INI [repeat any_token]
        deconstruct E1
```

```
        T1 [T]
    by
        INI [rT1 T1] [rT2 T1]
end function

function rT1 T0 [T]
    replace [repeat any_token]
        INI [repeat any_token]
    deconstruct T0
        F1 [F] * T1 [T]
    by
        INI [rF1 F1] [rF2 F1] [rT1 T1] [rT2 T1] [p *]
end function

function rT2 T1 [T]
    replace [repeat any_token]
        INI [repeat any_token]
    deconstruct T1
        F1 [F]
    by
        INI [rF1 F1] [rF2 F1]
end function

function rF1 F1 [F]
    replace [repeat any_token]
        INI [repeat any_token]
    deconstruct F1
        I1 [id]
    by
        INI [p I1]
end function

function rF2 F1 [F]
    replace [repeat any_token]
        INI [repeat any_token]
    deconstruct F1
        ( E1 [E] )
    by
        INI [rE1 E1] [rE2 E1]
end function
```

**One Grammar Method** It is possible to translate between two languages only transversing the parse tree and performing the identity transformation, while the target code is generated as a side effect. This method avoid minding with the grammar of the target language. The grammar is reduced to the grammar of the source language:

```
define program
    [E]
end define

define E
    [T] + [E] |
```

```
        [T]
end define

define T
        [F] * [T] |
        [F]
end define

define F
        [id] |
        ( [E] )
end define
```

In this method, it is necessary to write the target code in a file, otherwise the code would be lost.

```
external function msg M [any]
external function openFile nomeArq [any]
external function closeFile

function rE1
        replace [E]
                T1 [T] + E1 [E]
        by
                T1 [rT1] [rT2] + E1 [rE1] [rE2] [msg '+]
end function

function rE2
        replace [E]
                T1 [T]
        by
                T1 [rT1] [rT2]
end function

function rT1
        replace [T]
                F1 [F] * T1 [T]
        by
                F1 [rF1] [rF2] * T1 [rT1] [rT2] [msg '*]
end function

function rT2
        replace [T]
                F1 [F]
        by
                F1 [rF1] [rF2]
end function

function rF1
        replace [F]
                I1 [id]
        by
                I1 [msg I1]
```

```
end function

function rF2
        replace [F]
                ( E1 [E] )
        by
                ( E1 [rE1] [rE2] )
end function

function mainRule
        replace [program]
                E1 [E]
        by
                E1 [openFile 'output] [rE1] [rE2] [closeFile]
end function
```

## 4 Denotational Semantics

In order to be able to automatically translate code from source language to target language, it is necessary to specify the meaning of each structure of the source language. It is impossible for the computer (as it is for humans) to guess the meaning of a program in the source language only from its syntax. Denotational semantics is a mathematical formalism created by Strachey and Scott [SS71] to specify the meaning of programming languages. It aims at specifying the meaning of language constructs without explicit modeling computational processes. (As in operational semantics approach.)

How is it possible to comunicate the meaning of a word to a foreign person ? One way to communicate him/her the meaning is by pointing out the object denoted by the unknown word. If the word is "table", a table can be pointed out. Another way is to translate the word to a common language. The first way is the idea of denotational semantics, the meaning of a program is a computable function from its input to its output.

If the inputs range over the INPUT domain and the outputs range over OUTPUT, then to specify the meaning of a program is to give a denotation function, [[ ]]:program $\longrightarrow$(INPUT $\longrightarrow$ OUTPUT), that maps a program to its meaning, a function *meaning*: INPUT $\longrightarrow$ OUTPUT. It is, however, harder to point out a mathematical entity such as a computable function than to point out a concrete object. In practice, this mapping is defined by structural induction (cf. [Win93]). It is also necessary to use a language to write down the code of the meaning function; this language is usually a functional specification language based on $\lambda$-calculus. This will make a denotational specification look like a translation to the functional specification language. This fact will be used in subsection 5.4 in order to generate transformations out of the semantic specification.

### 4.1 Functional Specification Language

As mentioned before, the function that represents the meaning of a program must be coded in a functional language. This language is usually based on $\lambda$-calculus. The language used in this work will be called FSL (Functional Specification Language). FSL is a variation of $\lambda$-calculus where the most useful operators are predefined. A semantic domain is assigned to each $\lambda$-term; this domain information will be used in subsection 5.6 to generate code in C++ out of a $\lambda$-term.

Semantic domains are either primitive domains like booleans, strings, identifiers, and natural numbers, or are constructed from other domains by means of the operators: $\times$ (product), $+$ (coproduct), $\longrightarrow$ (exponential) and * (list). Example:

| Semantic Domains | |
|---|---|
| $\nu$ : N | natural numbers |
| $\sigma$ : M = Id $\times$ N | memory |
| $\phi$ : Fi = N* | files |
| $\gamma$ : Cf = Fi $\times$ M $\times$ Fi | configurations |

Besides the operations over the primitive domains, each domain constructing operator brings operations to build and decompose compound values. They are:

$\times$:
constructing operation:
$\ll$ , $\gg$ — if parameters are of type A and B, result is of type A $\times$ B.
decomposing operation:
$\downarrow$ 1 , $\downarrow$ 2 — projections on the first and second components.
$\downarrow$ 1: A $\times$ B $\longrightarrow$ A. $\downarrow$ 2: A $\times$ B $\longrightarrow$ B.
conditions:
$\downarrow 1 \circ \ll a , b \gg = a$
$\downarrow 2 \circ \ll a , b \gg = b$

+:
constructing operation:
a in C — where a $\in$ A and C = A + B, results c $\in$ C (injection).
b in C — where b $\in$ B and C = A + B, results c $\in$ C (injection).
decomposing operation:
c | A — where c $\in$ C and C = A + B, results a $\in$ A (coprojection).
c | B — where c $\in$ C and C = A + B, results b $\in$ B (coprojection).
testing operation:
c E A — where c $\in$ C and C = A + B, results p $\in$ BOOLEAN.
conditions:
( a in C ) | A = a
( b in C ) | B = b
( a in C ) E A = true and ( a in C ) E B = false
( b in C ) E B = true and ( b in C ) E A = false

$\longrightarrow$:
constructing operation ($\lambda$-abstraction):
$\lambda$a.b — where a $\in$ A and b $\in$ B, results f $\in$ A$\longrightarrow$B.
decomposing operation (function aplication):
Apply(f,a) — where f $\in$ A$\longrightarrow$B and a $\in$ A, results b $\in$ B.
conditions:
Apply($\lambda a_1$.b,$a_2$) = b[$a_2/a_1$]
Where b[$a_2/a_1$] is b where ever free ocurrence of $a_1$ is simultaneously replaced by $a_2$.

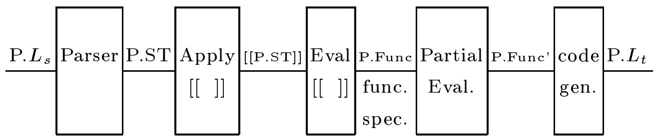
*:
constructing operation ($\lambda$-abstraction):
cons( , ): - takes values of type A and A*, result is of type A*.
decomposing operation:
head( ): A*$\longrightarrow$A and tail( ): A*$\longrightarrow$A*
conditions:
head(cons(a,l)) = a and tail(cons(a,l)) = l

## 5 STS — The Semantic Translation System

The approach proposed here consists of using [[ ]]$_{sl}$, the denotational specification of $L_s$, to extract a functional specification of the source code. This specification is then translated to an object oriented language to be executed.

### 5.1 Overview of STS

The STS way of translating $L_s$ into $L_t$ can be viewed as:

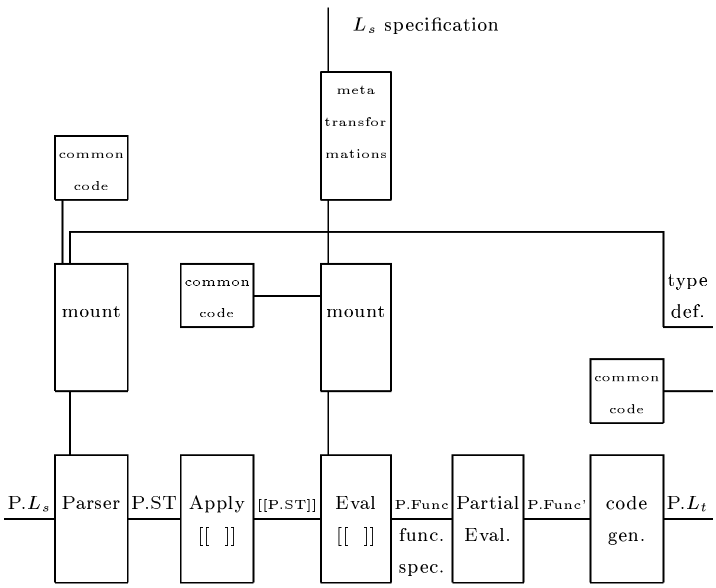



- P.$L_s$. Program written in the source language.
- Parser. As in the compiler, the parser can be generated from a syntatic specification of Ls.
- P.ST. Syntax tree of the same program in Ls.
- Apply [[ ]]. Formally applies the denotation function.

[ P.ST ]. Denotation function applied to its argument, but not yet evaluated.

- Evaluate [[ ]]. Transformations that evaluate the denotation function. Further we will see that these transformations are generated by the semantic specification.
- P.func. Functional specification of the program.
- Partial evaluation. Applies reductions to the functional specification. Compile time expressions can be solved in this phase.
- P.func'. Functional specification of the program after partial evaluation.
- Code generation. Translates the functional specification into an object oriented language.
- P.$L_t$. Program written in the target language.

Each of these steps handle program code, so they are suited to the transformational paradigm. Automatic generation of the language-dependent steps can be added to the previous diagram. Thus the new steps can be viewed as:




The new steps are:

**Mount** Each *mount* module is a different process. They are all called “mount” because their only tasks are to merge files, process macro substitutions, and pretty print the output. (so that it can be used as a transformational program)

**Meta-transformations** This is a major process in STS. From the syntatic part of the specification, it generates the parser (a transformational program). From the implementation of the semantic functions, it generates the evaluation transformations, which are another transformational program. Finally, from the signature of the semantic functions, it generates the type definitions. (an object oriented code, that is included in P.$L_t$ ) This process is also a transformation, which is the reason why they are called “meta-transformations”.

In STS, unlike the previous approaches, the semantics of the target program is given in a formal basis, written in a specification method with growing acceptance in the academic community. In writing the specification, one will focus only on the meaning of the constructions, no longer on control or data manipulation. The accuracy of the meaning of the target program depends solely on the accuracy of the denotational specification. The transformations, which embrace flow control and data manipulation, are correct by construction. One can say that the first scheme in this subsection works as a compiler from $L_s$ to $L_t$, whereas the second scheme works as a semantic-driven compiler generator, i.e., a full compiler generator, not a parser generator, like Yacc [Joh75] and many others. There are several reports on automatic semantic driven compiler generation. The first is [Mos75] (see also [Mos79]) followed by [JS80]. In the 1990's we distinguish [RFP94], [Mos94] and [Gue95]. [Jor92] generates a compiler by partial evaluating an $L_s$ interpreter, which can also be understood as a semantic specification.

### 5.2 The Specification Language

The STS system has two inputs: P.$L_s$, the program to be translated; and $[[\ \ ]]L_s$, the denotational specification of $L_s$. This subsection describes the syntax of the latter. Following [Gor79], [Pag81], a language specification in STS is divided in four parts:

- **Syntax.** Unlike most denotational specifications, STS requires the concrete syntax of the language. The reason is that STS must be able to extract enough information to generate a parser out of this part of the specification. We have thus chosen the TXL syntax to specify concrete syntax. (see TXL syntax in subsection 3.1)
- **Semantic domains.** Semantic domains are defined out of the basic domains BOOLEAN, INTEGER, NIL, STRING and the operators + (coproduct), x (product), $\longrightarrow$ (functional type or exponential) and * (list type or closure). Example:

| Semantic Domains | |
|---|---|
| $\upsilon$ : V = INTEGER + STRING | values |
| $\sigma$ : M = Id $\longrightarrow$ V | memory |
| $\phi$ : Fi = N* | files |
| $\gamma$ : Cf = Fi x M x Fi | configurations |

- **Semantic functions signature.** This part is syntatically embedded in the next one.
- **Semantic functions body.** This part allows for the use of pattern matching guards on the left side of the specification and an expression written in FSL (see subsection 4.1) on the right side. Example:

function Cs :: CMDS $\Longrightarrow$ CONF $\longrightarrow$ CONF ;
  Cs [[ [CMD] ; [CMDS] ]] ($\gamma$) =
  CONF : '{Apply} ( Cs : [[ CMDS ]] , CONF : '{Apply} ( C : [[ CMD ]] , $\gamma$ ) ) ;
end function

### 5.3 Generating the Parser

The transformations generated from the source language syntax act on the syntax tree of the program. Therefore it is necessary to perform syntax analysis on the input program before it is submitted to the transformation process. This process is also made automatic in STS. Generating a parser in TXL out of the source language syntax specification is a simple task. As we use the same syntax used in TXL to write syntax specifications, the first part of the parser in TXL is just a copy of the syntax specification written in the source language specification. The second part are rules that traverse the syntax tree, generating a written version of it. Example:

```
function rCMD1 X [CMD]
  replace [repeat anyToken]
    INI [repeat anyToken]
  deconstruct X
    'input VAR1 [ID]
  by
    INI [slice 'rCMD1] [slice ':(] [rID1 VAR1] [slice ')]
end function
```

There are meta-rules in the meta-transformation process that generate the rule above out of the grammar of the language.

### 5.4 The Semantic Metatransformations

This is an important step in STS, unlike other translating strategies, transformations are automatically built from $L_s$ semantic specification by meta-transformations. This step shows that a denotational specification already specifies a translation from the language being specified to $\lambda$-calculus. The semantic rule is translated to a reduction (transformation) that evaluates the denotation function. For example, the denotational rule:

$$[[CMD; CMDS]](\gamma) = [[CMD]] \circ [[CMDS]](\gamma) \tag{1}$$

is translated into a transformation rule:

$$[[CMD; CMDS]](\gamma) \triangleright [[CMD]] \circ [[CMDS]](\gamma) \tag{2}$$

There are, however, semantic equations that are not compositional, eg:

$$E[[E1 - E2]] = E[[E1 + (-E2)]] \tag{3}$$
$$C[repeatC]] = \ldots C[[C]] \ldots C[[repeatC]] \ldots \tag{4}$$

As pointed out by Mosses in [Mos83] these equations can be easily replaced by compositional ones (in the former case by macro-substitution, and in the latter case by the use of the fixed-point operator), thus defining a translation from $L_s$ to $\lambda$-calculus by structural induction. There are meta-rules in the meta-transformation process that generate the transformational rule out of the semantic functions. As the equation (1) and (2) are similar, we expect to obtain a simple meta-transformation. The following example shows the real code for equation (2):

```
rule rCMDS1
  replace [EXPR]
    '{CONF} ': '{Apply} '( Cs ': '[[ S [SYNTAX_EXPR] ']] ', '{CONF} ': gamma [UTE] ')
  deconstruct S
    'rCMDS1 ': '( CMD [SYNTAX_EXPR] , CMDS [SYNTAX_EXPR] ')
  by
    '{CONF} ': '{Apply} '( 'Cs ': [[ CMDS ]] ', 'CONF ': '{Apply} '( 'C ': [[ CMD ]] ', '{CONF} ': gamma ') ')
end rule
```

### 5.5 The Partial Evaluation Step

This step aims at resolving compile time expressions present in the functional specification. It uses transformation to implement the reduction relations of the language. Eg:

$$\text{Proj} ( \ll \text{EXP1} , \text{EXP2} \gg , 1 ) \rhd \text{EXP1}$$

Where $\ll$ , $\gg$ denotes a pair (product), Proj ( _ , 1) is the projection on the first component, and $\rhd$ is read as "reduces to". This reduction is implemented by the transformation:

```
rule projection1
  replace [EXPR]
    T1 [id] ': 'Proj '( T2 [id] ': '≪ E [EXPR] R[list_opt_rest_EXPR] '≫ ', N [integernumber] ')
  where
    N [ = 1 ]
  by
    E
end rule
```

### 5.6 Code Generation

This step makes the functional specification operational. As seen before the final code consists of three parts:

1. **Common Code**. These are C++ support libraries needed to compile the generated code. They include implementation of the basic types of the FSL: INTEGER, BOOLEAN and STRING, which are coded as C++ classes. They also include the implementation of FSL domain constructing operators: x, +, $\longrightarrow$, *. These operators are coded as C++ class templates. They are:
   - tPROD. This template implements x, the product domain constructing operator. Its prototype is:

     template <class T1, class T2> class tPROD;

     It offers the projection and pair constructing operators as methods. A ternary version of product is implemented by template tPROD3. Hence a domain like CONFIGURATIONS = FILE x MEMO x FILE can be directly translated into tPROD3<FILE,MEMO,FILE> instead of tPROD<tPROD<FILE,MEMO>,FILE>.
   - tCOPROD. This template implements +, the coproduct domain constructing operator. Its prototype is:

     template <class T1, class T2> class tCOPROD;

     It offers the coprojection and injection operators as methods.
   - tFUNC. This template implements $\longrightarrow$, the functional domain constructing operator. Its prototype is:

     template <class T1, class T2> class tFUNC;

     It offers the application and pointwise modification operators as methods. Since C++ is not a functional language, abstraction can not be a method, it is translated to a function definition.
   - tLIST. This template implements *, the list domain constructing operator. Its prototype is:

     template <class T> class tLIST;

     It offers the head( ) and tail( ) operators as methods, cons( ) is a friend generic function.

2. **Type Definition**. The semantic domain definitions in the denotational semantics of $L_s$ are used to create type declarations that are part of the target code. Example:
typedef tCOPROD<INTEGER,STRING> VALUE;
typedef tFUNC<ID,VALUE> MEMORY;
typedef tLIST<VALUE> ARQ;
typedef tPROD3<FILE,MEMORY,FILE> CONFIGURATION;

3. **P.**$L_t$. Input program written in C++. This is the product of transversing the functional specification of the input program. A new object is declared whenever necessary to store the value associated with a node. Example:
CONFIGURATION CONF1(&FILE1,&MEMO1,&FILE2);
FILE FILE3;
FILE3 = CONF1.proj1();
VALUE VALUE2;
VALUE2 = FILE3.head();

## 6 Consideration on the Target Code Correctness

One of the major advantages of the approach proposed in this article is the quality of the target code. Whereas in the usual transformational approach the correctness of the target code depends on the accuracy of the transformations, in STS it depends only on the correctness of $[[\ \_\ ]]._{sl}$, which is a more abstract specification of $L_s$, and less concerned with flow control or data manipulation. In this section, a more precise version of the statement above will be shown: "Whereas in a usual translating system it is necessary to prove its correctness on translating $L_s$ to $L_t$ for each $L_s$ and $L_t$, in STS a single correctness proof assures that the meaning of the input program is preserved for any $L_s$."

STS defines a translation function $STS(\_)$ from $L_s$ to C++. Given an input $P.L_s$, $STS(P.Ls)$ is the output provided by the STS system for this input. This translation is the composition of two functions:

– $T_{SL2FSL}$ is a function from L$_s$ to FSL. It is implemented by the "apply" and "evaluate" steps, introduced in subsection 5.1. This function is specific for one source language.
– $T_{FSL2C++}$ is a function from FSL to C++, defined by the "code generation" step. This translation is independent from the source language.

The composition of these functions is such that:

$$STS(P.L_s) = T_{FSL2C++}(T_{SL2FSL}(P.L_s)) \tag{5}$$

Two propositions — both independent from L$_s$ — are sufficient to assure that the translation STS( _ ) preserves the meaning of programs. The first states that the meta-transformations work properly, and the second states that the code generation is correct.

**Proposition 6.1** *The meta-transformations preserve the meaning of source language semantic specification, $[[\ \_\ ]]_{sl}$, ie:*

$$\forall\, L_s\, [[\_]]_{sl} = [[TXLRULES(L_s)]]_{TXL} \tag{6}$$

For a particular source language, $L_s$, the equation 6 becomes:

$$[[\_]]_{sl} = [[TXLRULES(L_s)]]_{TXL} \tag{7}$$

where TXLRULES($L_s$) are the TXL rules generated by the meta-transformations from $[[\ \_\ ]]_{sl}$. Note that executing TXL rules over a program in L$_s$ is the same as applying the translation $T_{SL2FSL}($ P ).

$$[[TXLRULES(L_s)]]_{TXL}(P.L_s) = T_{SL2FSL}(P.L_s) \tag{8}$$

**Proposition 6.2** *The code generation step preserves the meaning of programs written in the functional specification language, FSL.*

$$\forall\, P.FSL,\; P.FSL = [[T_{FSL2C++}(P.FSL)]]_{C++} \tag{9}$$

This result is proven by induction in P.FSL structure. As the code generation step translates from FSL to C++, a program in FSL is understood as its own semantic specification, whereas the semantic specification of a C++ program is given by $[[\ \_\ ]]_{C++}$.

This leads to:

$$\begin{array}{rl} & [[STS(P.L_s)]]_{C++} = \\ \text{by (5)} & [[T_{FSL2C++}(T_{SL2FSL}(P.L_s))]]_{C++} = \\ \text{by (8)} & [[T_{FSL2C++}([[TXLRULES(L_s)]]_{TXL}(P.L_s))]]_{C++} = \\ \text{by (7)} & [[T_{FSL2C++}([[P.L_s]]_{sl})]]_{C++} \end{array}$$

As $[[P.L_s]]_{sl}$ is a program written in FSL, equation 9 can be applied.

$$\begin{array}{rl} & [[STS(P.L_s)]]_{C++} = \\ & [[T_{FSL2C++}([[P.L_s]]_{sl})]]_{C++} = \\ \text{by (9)} & [[P.L_s]]_{sl} \end{array}$$

Hence it was proven that:

**Proposition 6.3** $STS(\_)$ *preserves the semantics of any input program written in any source language.*

## 7 Conclusion

There is a growing need for migrating legacy systems due to the fast development of new platforms. We discussed the usual solutions to this problem. A shortcoming of many existing solutions is that they lack a formal basis to assure the correctness of the target code. STS deals with this problem in such a way that the user of the system does not specify the translation by writing imperative programs or by writing transformations, but only the denotational specification of the source language. The transformations the system generates are correct by construction.

We expect that the use of a mapping from the functional specification language to an object oriented language that is similar to [Gue95] leads to an efficient target code. Preliminary results corroborate this assumption.

One may point out a drawback: there are two translations in this process - the first one from $L_s$ to a functional language, and the second from this to $L_t$. This may compromise the target code, for example, by making it resemble an interpreter. Further investigation of these consequences is relevant.

## References


[Boy89] J. Boyle. "Abstract Programming and Program Transformations - An Approach to Reusing Programs". *Software Reusability*, 1, 1989. Ed. Ted Biggerstaff, ACM Press.

[CC93] J. Cordy and I. Carmichael. "The TXL Programming Language Syntax and Informal Semantics". Technical report, Queen's University at Kinkston - Canada, 1993.

[CH96] Isabel Cafezeiro and Edward H. Haeusler. "Paradigmas de Linguagens de Programação: Uma Abordagem Geométrica". In *Anals of the 1st Brazilian Symposium of Programing Languages*, 1996.

[Gor79] M. Gordon. *"The Denotational Description of Programming Languages: An Introduction"*. Springer-Verlag, 1979.

[GS93] L.C. Guedes and A. Staa. "Um Processo de Reengenharia Econômico e Eficaz". In *Anais do VII Simpósio Brasileiro de Engenharia de Software*, 1993.

[Gue95] L.C. Guedes. *"Um Modelo Orientado a Objetos para Geracao Automatica de Compiladores"*. PhD thesis, Pontificia Universidade Católica, Rio de Janeiro, 1995.

[HHC93] A.M. Haeberer, E.H. Haeusler, and O.P. Coelho. "A Denotationo-Transformational Approach to the Solution of the Legacy Code Problem". Personal Notes, 1993.

[Joh75] S. C. Johnson. "yacc — Yet Another Compiler Compiler". Computing science technical report 32, AT&T Bell Laboratories, New Jersey, 1975.

[Jor92] J. Jorgensen. "Compiler Generation by Partial Evaluation". Master's thesis, University of Copenhagen, 1992.

[JS80] N.D. Jones and D.A. Schmidt. "Compiler Generation from Denotational Semantics". *LNCS*, 94:70–93, 1980. Springer-Verlag.

[LSA95] J. Leite, M. Sant'Anna, and Prado A. "Porting Cobol Programs Using a Transformational Approach". Monographs in computer science 39/95, Pontificia Universidade Católica, Rio de Janeiro, 1995.

[LSF94] J. Leite, M. Sant'Anna, and Freitas F. "A Technology Assembly for Domain Oriented Software Development". In *Proceedings of the 3rd International Workshop on Software Reuse*. IEEE Computer Society Press, 1994.

[Mos75] Peter D. Mosses. *"Mathematical Semantics and Compiler Generation"*. PhD thesis, Oxford University, 1975.

[Mos79] Peter D. Mosses. "SIS — Semantic Implementation System". Daimi md-30, Computer Science Department, Aarhus University, 1979.

[Mos83] Peter D. Mosses. "Abstract Semantic Algebras". In *Formal Description of Programming Concepts — II*. North-Holland Publishing Company, 1983.

[Mos94] Peter D. Mosses, editor. *"Proceedings of the First International Workshop on Action Semantics"*, Edinburgh, Scotland, 1994. North-Holland Publishing Company.

[Oli96] Alcione P. Oliveira. *"Casamento de Padrões em Ambientes para Processamento de Conhecimento"*. PhD thesis, Pontificia Universidade Católica, Rio de Janeiro, 1996.

[Pag81] Frank G. Pagan. *"Formal Specification of Programming Languages — A Panoramic Primer"*. Prentice-Hall, 1981.

[REF92] Reasoning Systems Incorporated, Palo Alto. *REFINE User's Guide*, 1992.

[RFP94] J. Ringstoerm, P. Ftritzson, and M. Pettersson. "Generating an Efficient Compiler for a Data Paralel Language from Denotational Specification". In *Proceedings of the 5th International Conference on Compiler Construction*, number 786 in LNCS, pages 248–262, Edimburg, April 1994. Springer Verlag.

[SS71] D.S. Scott and C. Strachey. "Toward a Mathematical Semantics for Computer Languages". Prg-6, Programming Research Group, Oxford University, 1971.

[Win93] Glynn Winskel. *"The Formal Semantics of Programming Languages — An Introduction"*. The MIT Press, 1993.