Proc. IEEE 1988 International Conference on Computer Languages, Miami, October 1988

TXL: A Rapid Prototyping System for Programming Language Dialects*

James R. Cordy
Department of Computing and Information Science
Queen's University at Kingston
Kingston, Canada K7L 3N6

Charles D. Halpern
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Eric Promislow
Department of Computing and Information Science
Queen's University at Kingston
Kingston, Canada K7L 3N6

Abstract

This paper describes a rapid prototyping system for extensions to an existing programming language. Such extensions might include new language features or might introduce notation specific to a particular problem domain. The system consists of a dialect description language used to specify the syntax and semantics of extensions, and a context sensitive syntactic transducer that automatically implements the extensions by transforming source programs written using them to equivalent programs in the original unextended language. Because the transformer is context sensitive, it is more powerful than traditional context free preprocessors and extensible languages and can be used to prototype language extensions involving significantly new programming paradigms such as object oriented programming.

Introduction

As the diversity of programming paradigms continues to grow and the importance of problem domain specific notation in programming languages is increasingly accepted [1], it becomes more and more important to be able to try out new language features and new notation. Ideally, we should be able to rapidly prototype the new language features in order to benefit from user experience before full scale production implementation and avoid expensive modifications to the language implementation later.

Because the expense of producing complete new language processors is prohibitive, the usual way of conducting such prototyping experiments involves implementing the new language features on top of an existing base language, creating a new *dialect* of the original base language. Traditionally, this has been done using either a regular or context-free syntactic preprocessor such as a macro processor, or by using an extensible programming language as the base language. These traditional solutions have several drawbacks.

* This work was supported by the Natural Sciences and Engineering Research Council of Canada.

Syntactic preprocessors such as the PL/I preprocessor [2], M4 [3] and the C preprocessor [4] generally limit the range of possible dialects to regular or context-free translations to the syntax of the original base language [5]. While this is a reasonably large set, it is by no means clear that all of the dialects we might wish to prototype fall in this class. In particular, dialects involving significantly new programming paradigms, such as object oriented and generic programming, cannot be prototyped in this way.

While the more powerful macro preprocessors and extensible languages such as ICON [6], CLEF [7] and Lithe [8] often allow a larger range of dialects than simply the context free set, they tend to place limits on the syntactic form of the dialect notation and remove that necessary degree of freedom in the prototyping capability. For example, macro preprocessors often limit the syntax of new constructs to simple variants of functional notation while extensible languages usually limit extensions to syntactic forms which are simple variants of the syntax of existing language features such as functional notation and binary operators.

By contrast, the *mkmac* extension tool for the language Scheme [9] provides the ability to add any desired syntax by giving an example of the syntax and explicitly specifying the transformation to Scheme as part of the macro definition. By taking advantage of the inherent self-reference capabilities afforded by its interpretive nature (Scheme is a variant of Lisp), significantly new language features can be added.

This paper describes TXL, a system explicitly designed to allow easy description and automatic prototype implementation of significantly new programming language features and programming paradigms. The goal of TXL is to provide an extension tool which allows some measure of the power and flexibility of *mkmac* for traditional Pascal-like compiled languages. TXL uses a context sensitive transformation algorithm that is not limited by the constraints typical of most other preprocessors and extensible languages, and is driven by a concise, readable dialect specification language that conveniently expresses the syntax and semantics of new language features.

TXL

Using the Turing programming language (or any other operational language) as a base, TXL provides the ability to describe the syntactic forms and run-time model of new language dialects at a very high level, and automatically provides an implementation of the new dialect. Dialects are described using a specially designed dialect description language (TXL).

Each dialect is described in two parts, the context-free syntactic forms of the dialect (described in terms of the syntactic forms of the base language using a BNF-like notation), and the run time model of the dialect (described in terms of a set of transformations to the base language). The TXL Processor uses these descriptions to transform source programs written in the described dialect to programs in the base language, which can then be compiled or interpreted by the normal base language processor (Figure 1).

The syntactic forms of the base language itself are described to TXL using the same BNF-like notation used to describe syntactic forms of the dialect. The base language syntactic description forms a data base of syntactic forms used to describe the syntactic structures of the dialect. For example, the syntactic forms of the Turing base include the forms *declarationsAndStatements, variableReference*, *assignmentStatement*, and so on.

The semantics of the dialect are described as a set of recursive context sensitive transformations from the syntactic structures of the dialect to generated base language structures.

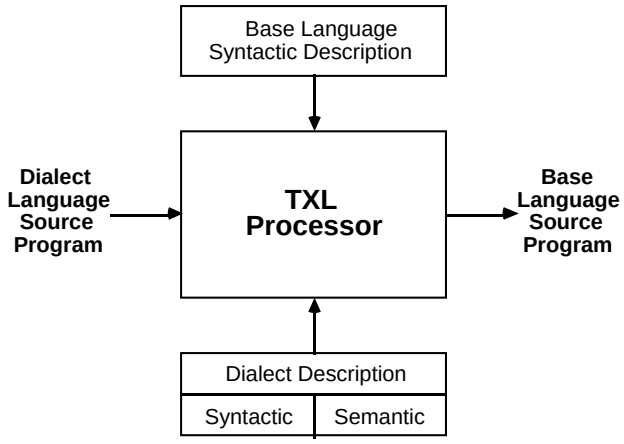

Figure 1. Dialect Descriptions for the TXL Processor

A Trivial Example

As a simple example of the description of a dialect, consider the addition of coalesced assignment short forms (i.e., the "+=", "–=" etc. of C) to the Turing language. The desired syntactic forms can be described in terms of the Turing base forms as a replacement of the *statement* syntactic form to include the original Turing statement forms plus a new form we call *coalescedAssignment* (Figure 2).

The new definition for the *statement* syntactic form replaces the original Turing form in the effective grammar of the dialect, so that the dialect accepted will include all of original Turing plus the new coalesced assignment statement. The form of the coalesced assignments themselves is described using the new syntactic form *coalescedAssignment* and its sub-form *coalescedOperator*.

```
% Trivial coalesced assignment dialect;
%   allows  a += b etc.

% Syntactic forms

define statement        % replaces Turing base
                        % syntactic form of same name
    choose
        [coalescedAssignment]    % new dialect
                                 %   statement form
        [assignment]             %   original Turing
        [assert]                 %   statement forms
        ...
        [get]
end define

define coalescedAssignment
    order
        [variableReference]
                [coalescedOperator]=  [expression]
end define

define coalescedOperator
    choose  +  -  *  /
end define
```

Figure 2. TXL Description of the Syntactic Forms of the Coalesced Assignment Dialect

Syntactic forms are described using a BNF-like notation in which the keyword ***order*** *indicates sequence and the keyword* ***choose*** *indicates alternation. The dialect syntactic forms are integrated into the base language grammar by replacing an existing base language syntactic form with a new form. In the above example, the new form of* statement *replaces the original Turing syntactic form of the same name in the dialect grammar.*

```
% Trivial coalesced assignment dialect (continued)

% Semantic transformations

rule replaceCoalescedOperators
    replace [statement]
        V [variableReference]
            Op [coalescedOperator]= E [expression]
    by
        V := V Op ( E )
end rule
```

Figure 3. TXL Description of the Semantic Transforms of the Coalesced Assignment Dialect

The semantics of the dialect are described using a set of rules that transform the syntactic forms of the dialect to semantically equivalent base language structures. In this case, every occurrence of a statement containing the dialect syntactic form coalescedOperator *is transformed to an assignment statement using the corresponding Turing operator.*

The meaning of the new syntactic form is described as a transformation to equivalent Turing base language code. In this case, for example, the transformation changes the coalesced assignment *a += b* to the semantically equivalent Turing statement *a := a + b* (Figure 3).

Implementation of TXL

The TXL processor consists of three parts, the Parser, the Transformer and the Deparser (Figure 4). The TXL parser merges the base language syntactic description and user-supplied dialect syntactic description to form an integrated dialect language syntactic description. The merge is done by simply replacing each syntactic form specification (i.e., production) of the base language grammar by the dialect syntactic form specification of the same name (if any). In this way, the syntax of new dialect features is smoothly integrated into the features of the original base language. Using this integrated grammar of the dialect language, the parser reads in dialect language source programs and transforms them into dialect language parse trees that can be manipulated by the transformer.

As an example, the syntax of the coalesced assignment dialect of the Turing base was specified by simply copying the existing Turing base language *statement* syntactic form into the dialect description and adding an alternative for the new syntactic form *coalescedAssignment*. The dialect's new *statement* syntactic form then replaced the original Turing form in the integrated dialect grammar, effectively adding coalesced assignments to the dialect language syntax.

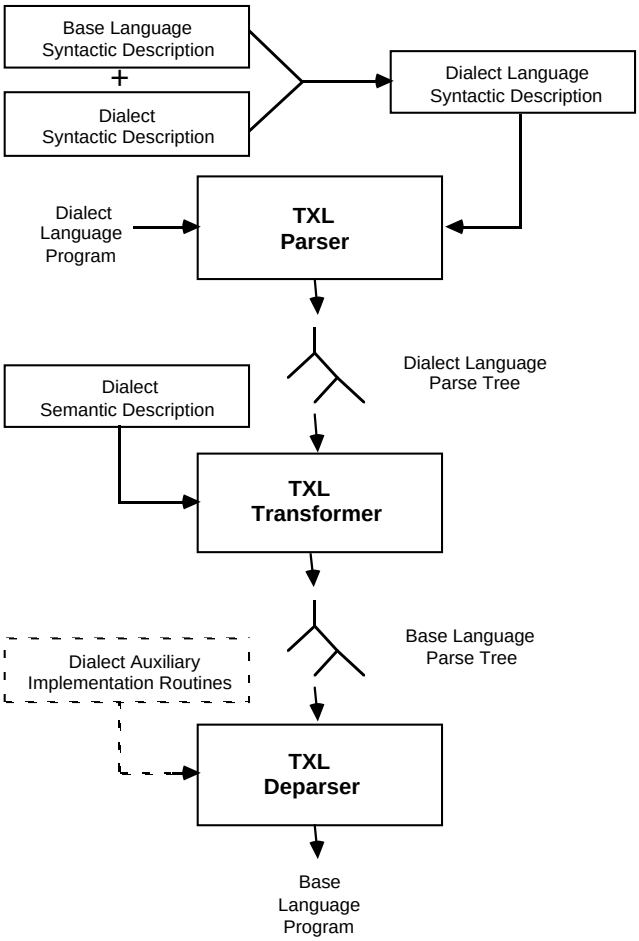

Figure 4. Implementation of the TXL Processor

The TXL transformer uses the dialect language semantic transform rules to recursively transform the dialect program parse tree to a parse tree for a base language program with equivalent semantics. The transform is done using a general purpose tree pattern matching algorithm. Beginning with the main (first) transformation rule in the dialect semantic description, the algorithm searches the parse tree for instances of the rule's anchor node which match the rule's pattern and replaces the subtree of each matched instance with a new subtree for the replacement. Other transformation rules may then be applied to the replacement subtree in a similar fashion, and so on, recursively applying replacements down the tree.

As an example, the main transformation rule of the coalesced assignment dialect (figure 3) specifies that in each subtree below a *statement* node, any subtree matching the pattern *variableReference coalescedOperator = expression* should be replaced by another *statement* subtree containing the assignment statement *V := V Op (E)* where *V, Op* and *E* are the original

subtrees for the *variableReference, coalescedOperator* and *expression* matched by the pattern. In order to maintain the structural integrity of the parse tree throughout the process of transformation and allow recursively applied transforms, the replacement subtree is re-parsed using the anchor node production of the dialect grammar before being linked in.

Finally, the TXL deparser generates the final base language source program by walking the base language parse tree resulting from the transformations using a leftmost depth-first traversal of the tree. Some dialects, such as those introducing concurrency primitives, may involve inclusion of auxiliary implementation routines from a library in the generated result as well.

A More Challenging Example

One common modern programming technique not present in the Turing language is the ability to declare generic (i.e., type parameterized) procedures and functions. An obvious Turing dialect then is one which has this feature. However, with TXL it is just as easy to describe a dialect that alallows not just gener procedures and functions, but arbitrary generic declarations including generic modules, procedures, functions, variables and types. We could imagine using such a facility to declare generics for classic data structures such as stacks, for example :

```
generic SimpleStack (someSize, someType)
    type SimpleStack :
        record
            depth : 0 .. someSize
            contents : array 1 .. someSize
                              of someType
        end record
```

Later in the generic dialect program, we might instantiate a stack or two :

```
% Instantiate and use a type for big
% stacks of strings
const bigDepth := 100
instance bigStackOfString :
             Stack (bigDepth, string)
var bs1, bs2 : bigStackOfString
% Initialize stacks bs1 and bs2
bs1.depth := 0
bs2.depth := 0
% Push the string "hi there" on bs2
bs2.depth := 1
bs2.contents (bs2.depth) := "hi there"
% Assign the entire value of bs2 to bs1
bs1 := bs2
```

The TXL description of this dialect is given in Figure 5. Two syntactic forms are added to the Turing declaration forms. The *genericDeclaration* form allows any form of declaration in the dialect (including generic declarations themselves) to be made generic. The *instanceDeclaration* form allows instances of any such

```
% Generic dialect of Turing;
%   allows arbitrary generic declarations

% Syntactic forms

define declaration
    choose
        [genericDeclaration]        % new dialect
                                %   declaration form
        [instanceDeclaration]       % new dialect
                                %   declaration form
        [constantDeclaration]       % original Turing
        [typeDeclaration]           %  declaration
forms
        ...
        [moduleDeclaration]
end define

define genericDeclaration
    order
        generic [id] ( [list id] )
            [declaration]
end define

define instanceDeclaration
    order
        instance [id] : [id] ( [list id] )
end define

% Semantic transformations

rule replaceGenerics
    replace [declarationsAndStatements]
        generic Gname [id] ( Formals [list id] )
            Decl [declaration]
        RestOfScope [declarationsAndStatements]
    by
        RestOfScope
            [@fixInstantiations Gname Formals Decl]
end rule

rule fixInstantiations Gname Formals Decl
    replace [declaration]
        instance Iname [id] : Gname ( Actuals [list id]
)
    by
        Decl    [@simpleSubst Gname Iname]
            [@simpleSubst Formals Actuals]
end rule

rule simpleSubst Old New
    replace Old by New
end rule
```

Figure 5. TXL Description of the Generalized Generic Dialect of Turing

generic declarations to be instantiated. The intended semantics is that each instance of a generic declaration declares a new object of the original generic object type, for example, an instance of a generic type declaration has the effect of a type declaration, an instance of a generic procedure declaration has the effect of a procedure declaration, and so on.

The semantic transformations of the dialect describe this semantics as follows. The main rule *replaceGenerics* searches in the parse tree for occurrences of the syntactic form *declarationsAndStatements* (i.e., the body of a Turing language scope), and within each such occurrence (i.e., scope) finds each generic declaration. It then replaces the remainder of the scope of the generic declaration by the same scope with the generic declaration removed and the instances of the generic replaced by instantiations of the generic. The actual replacement of instances with instantiations of the body is achieved by the second transformation rule, *fixInstantiations*.

Given as parameters the name of a declared generic, its formal parameter list and its body declaration, the *fixInstantiations* rule replaces each declaration of an instance of the generic in the scope of its application with a copy of the generic's body in which the name of the generic declaration is replaced by the name of the instance declaration and the formal parameter names of the generic have been replaced by the actual parameters given in the instance. Both the substitution of the instance name for the generic name and the substitution of the actuals for the formals is achieved by the last transformation rule, *simpleSubst*. This rule simply replaces each item in its first parameter (which may be a list) by the corresponding item in its second parameter over its range of application.

As an example of the kind of transformation done by TXL on a source program of the generic dialect, consider the *bigStackOfString* example given earlier. Given the TXL specification of Figure 5 and the *bigStackOfString* example as input, the TXL processor would output the Turing language result :

```
% Instantiate and use a type for big
% stacks of strings
const bigDepth := 100
type bigStackOfString :
    record
        depth : 0..bigDepth
        contents :
            array 1..bigDepth of string
    end record
var bs1, bs2 : bigStackOfString
```

Of course, a more reasonable generic characterization of the stack data structure would be as an abstract data type complete with the operations *Push*, *Pop* and *Top*. Because our generic dialect allows generics of any declaration, we can also do this within the dialect by making a generic module :

```
generic Stack (someSize, someType)
    module Stack
        export (Push, Pop, Top)

        var depth : 0..someSize := 0
        var contents :
            array 1..someSize of someType

        procedure Push (element: someType)
            pre depth < someSize
            depth := depth + 1
            contents (depth) := element
        end Push

        function Top : someType
            pre depth > 0
            result contents (depth)
        end Top

        procedure Pop
            pre depth > 0
            depth := depth - 1
        end Pop
    end Stack
```

Instances of the generic module *Stack* would then themselves be modules with the operations *StackInstance. Push, StackInstance.Pop* and *StackInstance.Top*.

Scope and Limitations of the Technique

The scope of possible transformations far exceeds the examples shown above. TXL is capable of arbitrary general pattern matching, recursive transformations, arbitrary code motion, generation of unique new identifiers and reference to auxiliary support routines. It has been used to specify and implement several dialects of the Turing programming language including a complex arithmetic dialect, an object oriented programming dialect, a SNOBOL-like pattern matching dialect and a concurrent programming dialect with processes and monitors.

The Turing programming language is particularly well suited as a base language for dialects because of its relative lack of syntactic restrictions (in particular, its lack of ordering restrictions on declarations and statements), its value-inherited type inference and its ability to reference external routines. While the lack of any of these base language features would not in theory restrict the range of dialects that can be described, in practice
they do help to keep the descriptions of new dialects elegant, concise and readable.

The range of dialects that can be described using TXL is restricted to some extent by the power of the base language chosen. For example, using Turing or any other Pascal-like programming language as a base does not allow us to describe a dialect containing the paradigm of program self-reference, because no mapping to base language source can successfully introduce that new concept into the execution of the resulting base language program.

TXL is most suitable for rapid prototyping of new programming constructs, notations and dialects. As demonstrated by the examples in this paper, it is relatively easy to make working transformations for new features whose semantics are well understood as run time models, but it is very difficult to make efficient ones and that task is best left to professional programming language implementors.

Acknowledgements

The TXL technique and dialect specification language were designed by C.D. Halpern and J.R. Cordy at the University of Toronto [10]. The TXL processor implementation was prototyped by C.D. Halpern at the University of Toronto and refined to production by E. Promislow at Queen's University [11]. The Turing programming language [12] [13] was designed and implemented by R.C. Holt, J.R. Cordy, M.P. Mendell, S.G. Perelgut and others at the University of Toronto. The development of TXL was generously supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. B. Hailpern, "Multiparadigm Research: A Survey of Nine Projects", IEEE Software 3,1 (January 1986).

2. OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, Form No. GC33-0009-4, IBM Data Processing Division, 1976.

3. B.W. Kernighan and D.M. Ritchie, "The M4 Macro Processor", in the UNIX Programmer's Manual, Seventh Edition, January 1979.

4. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.

5. T.A. Standish, "Extensibility in Programming Language Design", Proc. 1975 Spring Joint Computing Conference, AFIPS 44, 1975.

6. R.E. Griswold and M.T. Griswold, *The ICON Programming Language*, Prentice- Hall 1983.

7. J.M. Triance and P.J. Layzell, "CLEF - A COBOL Language Enhancement Facility", Computation Dept. Report 273, University of Manchester Institute of Science and Technology, December 1982.

8. D. Sandberg, "Lithe: A Language Combining a Flexible Syntax and Classes", Proc. 9th ACM Symposium on Principles of Programming Languages, Jan. 1982.

9. E. Kohlbecker, "Using mkmac", Technical Report 157, Computer Science Department, Indiana University, May 1984.

10. C.D. Halpern, "TXL: A Rapid Prototyping Tool for Programming Language Design", M.Sc. thesis, Department of Computer Science, University of Toronto, January 1986.

11. E. Promislow, "Semantic Transformations with Syntactic Constructs: A Run-time Model for the Turing Extender", M.Sc. thesis, Department of Computing and Information Science, Queen's University at Kingston, expected 1988.

12. R.C. Holt and J.R. Cordy, "The Turing Language Report", Technical Report CSRI-153, Computer Systems Research Institute, University of Toronto, December 1983.

13. R.C. Holt, P.A. Matthews, J.A. Rosselet and J.R. Cordy, *The Turing Programming Language: Design and Definition*, Prentice-Hall 1988.