Proc. IEEE International Conference on Computer Languages, New Orleans, March 1990.

Specification and Automatic Prototype Implementation of Polymorphic Objects in TURING Using the TXL Dialect Processor*

James R. Cordy
Eric Promislow

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

Abstract

Object-oriented dialects of existing programming languages are often implemented using a preprocessor that translates from the dialect to an equivalent program in the original programming language. Unfortunately, the nature of the preprocessing done by these implementations is hidden in the ad-hoc algorithms of the preprocessors themselves except as demonstrated by examples. This paper describes an attempt to catalogue and generalize these syntactic transformations using a simple set of applicative transformation rules expressed in the TXL dialect description language. Example transformation rules for implementing object types and parametric polymorphism in an object-oriented dialect of the Turing programming language are given in the paper. These rules easily generalize to other languages of the Pascal family and have been used to automatically implement Objective Turing.

Introduction

Object-oriented dialects of existing programming languages such as C are commonly prototyped using a preprocessor that translates from the object-oriented dialect to the base language. This was the case, for example, for C++ [1] and Objective C [2]. However, the nature of the preprocessing done by these implementations is often hidden in the ad-hoc algorithms of the preprocessors themselves, and is at best exposed by a few simple examples of the results of the preprocessing.

The Objective Turing project is an attempt to rectify this situation by explicitly specifying each key feature of object-oriented programming as a context-sensitive syntactic transform to the (non-objective) Turing programming language [3] using the TXL dialect specification language [4]. By specifying each feature using an independent TXL transformation rule which clearly and compactly encodes the necessary syntactic transforms, we can compile a reference catalog for preprocessor implementation of object-oriented features that can be used to guide future preprocessor implementations for other languages.

This paper introduces the TXL dialect specification language and gives a complete set of applicative transformation rules to implement polymorphic object types as a dialect of Turing. These rules have been used to automatically implement Objective TURING using the TXL dialect processor.

The TXL Dialect Processor

TXL, the Turing eXtender Language [4] is a system designed to allow easy description and automatic prototype implementation of new programming language features as dialects of an existing programming language such as Turing. The goal of TXL is to provide some measure of the power and flexibility of interpretive extensible languages for traditional Pascal-like compiled languages. TXL uses a context-sensitive syntactic transformation algorithm that is not limited by the constraints typical of 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.

Using Turing (or any other language) as a base, TXL provides the ability to describe new language dialects at a very high level, and automatically provides prototype implementations. Each dialect is described in two parts, the context-free syntactic forms (described in terms of the syntactic forms of the base language using a BNF-like notation), and the run time model of the dialect (expressed as a set of applicative syntactic transformations to the base language). The TXL Processor uses these descriptions to automatically transform dialect programs to programs in the base language.

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

The syntactic forms of the base language are described using the same BNF-like notation used to describe syntactic forms of the dialect. These base language syntactic forms serve as a data base of syntactic forms used to describe the syntax of the dialect. For example, the syntactic forms of the Turing language base include the forms *declarationsAndStatements, variableReference, assignment,* 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 base language structures.

As a simple example 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 statements plus a new form we call *coalescedAssignment* (Figure 1).

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

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

Transformation is achieved by applying the transformation rules to the abstract syntax tree of each dialect language program. For example, the main transformation rule of the coalesced assignment dialect (figure 1) 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.

Objective TURING

Turing [3] is a new general purpose programming language in the tradition of Pascal. In addition to the features provided by Pascal, Turing provides other modern programming features such as encapsulation using modules, varying-length character strings, type-safe variant records, safe dynamic storage and pointers, parametric procedures, and run-time constants. Turing does not however provide any object-oriented features such as object types, polymorphism, inheritance or dynamic binding.

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

% Part 1:  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


%  Part 2:  Semantic transformations

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

Figure 1. TXL Description of a Simple Coalesced Assignment Dialect.

In Part 1, the syntactic forms of the dialect 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.*

In Part 2, 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.*

Modules and Information Hiding

One feature of object-oriented systems that the Turing language already provides is information hiding in the form of *modules.* Turing modules collect a set of data structures and the procedures that manipulate them into a single opaque package. Only the *exported* attributes of the module can be accessed from outside it. From an object-oriented point of view, a Turing module is a single instance object, and its exported procedures are the methods of the object. Figure 2 shows an example of a Turing *stack* module.

Although we will use them as a vehicle to implement object types in this paper, modules are not a prerequisite for the transformations presented. Because the information hiding provided by modules can itself be expressed as a syntactic transform to procedures and records [5], our transformations are easily extended to apply to languages without modules such as Pascal.

```
module stack
    import (error)
    export (push, pop)

    const maxdepth := 100
    var storage : array 1 .. maxdepth of int
    var depth : 0 .. maxdepth

    procedure push (e : int)
        if depth <= maxdepth then
            depth := depth + 1
            storage (depth) := e
        else
            error ("stack overflow")
        end if
    end push

    procedure pop (var e : int)
        if depth > 0 then
            e := storage (depth)
            depth := depth - 1
        else
            error ("stack underflow")
        end if
    end pop

    depth := 0
end stack
```

Figure 2. A Stack Module in Turing.

Object Types

The distinction between Turing modules and true objects is that modules are not first-class types and can not be used as the type of variables, parameters and so on. The obvious way to add objects to Turing, then, is to allow modules to be types. Using such an extension, the *stack* module could be used as the body of a type declaration and the resulting type used to declare instances (figure 3).

Since we intend to extend rather than modify Turing, our syntax will explicitly use the keyword *object* in place of *module* for module types. This additional syntactic form can be expressed in TXL by replacing the Turing base syntactic form *typeSpec* with an extension that adds object types :

```
define typeSpec
    choose
        [objectType]        % added new syntactic form

        [standardType]      % original Turing
        [arrayType]         %   type spec syntactic forms
            . . .
        [namedType]
end define

define objectType
    order
        object
            [importList]
            [exportList]
            [moduleBody]
        'end [id]
end define
```

The syntactic forms *importList, exportList* and *moduleBody* used to specify the *objectType* syntactic form are inherited from the Turing base grammar.

This syntactic specification is deceptively simple. However, its semantics have no direct reflection in Turing, and the problem of transforming *objectTypes* into equivalent Turing programs will involve several steps, each specified using a TXL transformation rule. Overall, the transformation we will be specifying works as follows:

Every declaration of an *objectType* in the dialect program will be transformed into a Turing module with the same body. The internal variables of the object type will be gathered together into a data record type that will be exported from the module. Global (initializing) statements of the object type will be gathered into an initializing procedure for object data records that will also be exported. Each exported procedure (method) of the object type will be transformed to take an extra parameter of the data record type, representing the private data of the calling object instance.

```
type stack :
    object
        import (error)
        export  (push, pop)

        . . . same body as the Stack
                module of figure 2  . . .

    end stack

var stack1 : stack
var stack2 : stack

stack1.push (5)
stack1.push (7)

stack2 := stack1            % assign entire stack1 object
                            % to stack2
var x : int
stack2.pop (x)
assert  x = 7
```

Figure 3. A Stack Object Type in the Objective Turing Dialect and Example Instances.

In the scope of the object type's declaration, each variable declaration of the object type will be replaced by a variable declaration of the exported data record type followed by a call to the initializing procedure. Finally, all calls to procedures (methods) of the variable instance will be transformed into calls to the corresponding procedures of the module, passing the instance's data record as an extra argument.

The transformation is governed by a *mainRule* that simply serves to get things started, since for efficiency reasons TXL limits the extent of a rule's pattern search to a specified scope of the abstract syntax tree. This main rule simply says that the rule *fixObjects* should be applied to every scope of the program.

```
rule mainRule
    replace [declarationsAndStatements]
        P [declarationsAndStatements]
    by [declarationsAndStatements]
        P [fixObjects]
end rule
```

Step 1. Convert Object Types to Modules.

The rule *fixObejcts* syntactically converts each object type declaration to a Turing module and specifies the scopes in which several sub-rules are to be applied.

```
rule fixObjects
    replace [declarationsAndStatements]

        type ObName [id] :
            object
                ObImport [importList]
                ObExport [exportList]
                ObBody [declarationsAndStatements]
            'end ObName
        RestOfScope [declarationsAndStatements]

    by [declarationsAndStatements]

        module ObName
            ObImport
            ObExport  [addObjectAndInitializerExport]
            ObBody
                [sortDeclarationsAndStatements]
                [makeObjectRecordTypeAndEnterFields]
                [makeObjectInitializerProcedureAndEnterStatements]
                [addObjectParameterToProcedures]
        'end ObName
        RestOfScope
                [transformObjectReferences ObName]
end rule
```

The rule states that in each scope containing a type declaration of an *objectType,* the declaration should be replaced by a module with the same import list, export list and body as modified by several other rules, and that the rule *transformObjectReferences* should be applied to the scope of the type declaration itself.

Step 2. Add Exported Names for the Object Data Record Type and the Object Initializer Procedure.

The rule *addObjectAndInitializerExport* adds two new names to the list of identifiers exported from the module: the name of the object data record type, *DataRecordType,* and the name of the object data record initializer procedure, *InitializeDataRecord.* These names need not be unique since they are hidden inside a Turing module. If unique names were needed, the *gensym* TXL primitive would be used to generate them.

```
rule addObjectAndInitializerExport
    replace [exportList]
        export ( First [id]  Rest [repCommaId] )
    by [exportList]
        export ( DataRecordType,
                InitializeDataRecord,  First  Rest )
end rule
```

The missing comma between *First* and *Rest* in the export list is not an error. The Turing base grammar uses the recursive production *repCommaId* to specify any number of repetitions (including zero) of a comma followed by an identifier. Because application of this rule is bound (by the *fixObjects* rule) to the export list of the object module, it will match only that one export list.

Step 3. Sort the Declarations and Statements in the Object Module.

The Turing programming language allows interspersing of declarations and statements in a scope. In particular, variable declarations, procedures and initializing statements in a module body may be arbitrarily intermixed. In order to create a record type containing all the private variables of the object, and the initializing procedure for object data records, the module body must be re-ordered to gather all variable declarations and all initializing statements together. (This does not change the semantics of the module.) The following rules specify the re-ordering.

```
rule sortDeclarationsAndStatements
    replace [declarationsAndStatements]
        ObBody [declarationsAndStatements]
    by [declarationsAndStatements]
        ObBody
            [sortDS]    % declarations before statements
            [sortTV]    % constants and types before
                        %       variables and procedures
            [sortVP]    % then variables, then procs
end rule

rule sortDS
    replace [declarationsAndStatements]
        S [statement]
        D [declaration]
        R [declarationsAndStatements]
    by [declarationsAndStatements]
        D
        S
        R
end rule

rule sortTV
    replace [declarationsAndStatements]
        V [variableOrSubprogramDeclaration]
        T [constantOrTypeDeclaration]
        R [declarationsAndStatements]
    by [declarationsAndStatements]
        T
        V
        R
end rule

rule sortVP
    replace [declarationsAndStatements]
        P [subprogramDeclaration]
        V [variableDeclaration]
        R [declarationsAndStatements]
    by [declarationsAndStatements]
        V
        P
        R
end rule
```

Each of these sub-rules specifies a bubble sort that bubbles up instances of one kind of declaration or statement before another. For example, the *sortDS* rule specifies that every occurrence of the pair (*statement, declaration*) should be replaced by the same pair in reverse order. This rule is repeatedly applied until there are no more occurrences of the misordered pair it is looking for in its scope of application (the body of the object module). Although it is tempting to think of TXL rules as algorithmic, they are in fact applicative and can be applied any time their pattern matches any subtree in their scope of application.

Step 4. Gather the Object Module's Private Variables into the Object Data Record Type.

As the object module's variable declarations are grouped together by the sorting rules, they can be gathered into a new record type. This type will be used as the type of an additional parameter to each procedure of the module giving the data fields of the object instance associated with each call. The transform has two parts: first a new empty record type declaration is inserted before the first variable in the scope, then each variable declaration is moved in as a field of the record type. These two steps will necessarily happen in sequence since the pattern of the second rule will never match until the first rule has been successfully applied.

```
rule makeObjectRecordTypeAndEnterFields
  replace [declarationsAndStatements]
      ObBody [declarationsAndStatements]
  by [declarationsAndStatements]
      ObBody
          [makeObjectRecordType]
          [enterObjectRecordTypeFields]
end rule

rule makeObjectRecordType
  replace [declarationsAndStatements]
      V  [variableDeclaration]
      Rest  [declarationsAndStatements]
  by [declarationsAndStatements]
```

```
      type DataRecordType :
          record
          'end record
      V
      Rest
end rule

rule enterObjectRecordTypeFields
  replace [declarationsAndStatements]
      type DataRecordType :
          record
              R [repRecordField]
          'end record
      var V [id] : T [typeSpec]
      RestOfScope [declarationsAndStatements]

  by [declarationsAndStatements]
      type DataRecordType :
          record
              V : T
              R
          'end record
      RestOfScope  [fixObjectVariableReferences V]
end rule
```

Step 5. Change References to the Object's Private Variables to Reference the Data Record Parameter of the Object Procedures.

As each private variable is moved into the object's data record type by the *enterObjectRecordTypeFields* rule, all internal references to it are changed to refer to the corresponding field of the data record parameter of the procedure containing the reference. This rule is parameterized by the identifier of the variable whose references are being transformed.

```
rule fixObjectVariableReferences Var [id]
  replace [id]
      Var
  by [reference]
      DataRecord . Var
end rule
```

Step 6. Gather the Object Module's Initializing Statements into the Object Data Record Initializer Procedure.

Since the sorting rules gather the initializing statements together at the end of the module's scope, creating the initializer procedure is relatively straightforward. The rule *makeObjectInitializerProcedureAndEnterStatements* simply grabs everything from the first statement to the end of the sorted body and puts it in a new procedure named *InitializeDataRecord.*

```
rule makeObjectInitializerProcedureAndEnterStatements
  replace [declarationsAndStatements]
      P [subprogramDeclaration]
      S [statement]
      Rest [declarationsAndStatements]
  by [declarationsAndStatements]
      P
      procedure InitializeDataRecord
                  (var DataRecord : DataRecordType)
          S
          Rest
      'end InitializeDataRecord
end rule
```

Step 7. Add an Object Data Record Parameter to Each Procedure of the Module.

This rule adds an additional object data record type parameter called *DataRecord* as the first parameter to each procedure of the object module. Nullary procedures, which for brevity are not handled here, would require a slightly different rule to be applied in addition to this one.

```
rule addObjectParameterToProcedures
  replace [declarationsAndStatements]

      procedure PName [id]
              ( Arg1 [parameterDeclaration]
                RestOfArgs [repCommaParameterDecl] )
          PBody [subprogramBody]
      procedure InitializeDataRecord
              InitPList [optParameterList]
          IBody [subprogramBody]
      RestOfScope [declarationsAndStatements]

  by [declarationsAndStatements]

      procedure InitializeDataRecord  InitPList
          IBody
      procedure PName
              ( var DataRecord : DataRecordType,
                      Arg1  RestOfArgs )
          PBody
      RestOfScope
end rule
```

In order to avoid an infinite sequence of matches, the *addObjectParameterToProcedures* rule keeps track of which procedures it has already transformed by moving each one below the *InitializeDataRecord* procedure after it has been done, and matching only the procedure immediately above it.

Step 8. Transform Declarations of Instances of the Object Type into Declarations of Object Data Records.

The rule *transformObjectReferences* handles the final two steps of the transform. The first of these is the change of declarations of instance variables of the object type into declarations of variables of the *DataRecordType* exported by the object module followed by a call to the *InitializeDataRecord* procedure of the module.

```
rule transformObjectReferences  ObName [id]
  replace [declarationsAndStatements]
      var ObVar [id] : ObName
      RestOfScope [declarationsAndStatements]
  by [declarationsAndStatements]
      var ObVar : ObName . DataRecordType
      ObName . InitializeDataRecord (ObVar)
      RestOfScope [changeObjectProcedureCalls
                                  ObVar  ObName]
end rule
```

Step 9. Change Calls to the Object Instance's Procedures into Calls to the Object Module.

The second step is the *changeObjectProcedureCalls* rule, applied by the *transformObjectReferences* rule to the scope of each object instance variable. This rule takes two TXL parameters, the name of the instance variable and the name of the object type. The *repCommaExpn* production is similar to the *repCommaId* production explained earlier, and includes the leading comma in the rest of the argument list to the procedure (if any).

```
rule changeObjectProcedureCalls ObVar [id] ObName [id]
  replace [procedureCall]
      ObVar . PName [id]  ( FirstArg [expn]
                  RestOfArgs [repCommaExpn] )
  by [procedureCall]
      ObName . PName (ObVar, FirstArg
                                  RestOfArgs)
end rule
```

Figure 4 shows the result of applying the entire transform to the example stack object type of figure 3.

Parametric Polymorphism

Most proponents of the object-oriented paradigm consider some kind of polymorphic capability to be essential. While true dynamic polymorphism might be desirable in some contexts, many systems bow to the requirements of efficient compilability and settle for some lesser form, such as static parametric polymorphism in the sense of [6] instead. C++, for example, limits its

```
module stack
  import (error)
  export (DataRecordType, InitializeDataRecord,
            push, pop)

  const maxdepth := 100

  type DataRecordType:
      record
          storage : array 1 .. maxdepth of int
          depth : 0 .. maxdepth
      end record

  procedure InitializeDataRecord (var DataRecord :
                                      DataRecordType)
      DataRecord.depth := 0
  end InitializeDataRecord

  procedure push (var DataRecord : DataRecordType,
                                      e : int)
      if DataRecord.depth <= maxdepth then
          DataRecord.depth := DataRecord.depth + 1
          DataRecord.storage (DataRecord.depth) := e
      else
          error ("stack overflow")
      end if
  end push

  procedure pop (var DataRecord : DataRecordType,
                                      var e : int)
      if DataRecord.depth > 0 then
          e := DataRecord.storage (DataRecord.depth)
          DataRecord.depth := DataRecord.depth - 1
      else
          error ("stack underflow")
      end if
  end pop
end stack

var stack1 : stack.DataRecordType
stack.InitializeDataRecord (stack1)
var stack2 : stack.DataRecordType
stack.InitializeDataRecord (stack2)

stack.push (stack1, 5)
stack.push (stack1, 7)

stack2 := stack1              % assign entire stack1
object
                              % to stack2
var x : int
stack.pop (stack2, x)
assert  x = 7
```

Figure 4. The Result of Transforming the Stack Object Type of Figure 3 to a Turing Module Using the Rules of the Object Type Dialect.

```
type class  stack (maxdepth, elementType) :
  object
      import (error)
      export (push, pop)

      var storage :
          array 1 .. maxdepth of elementType
      var depth : 0 .. maxdepth

      procedure push (e : elementType)
          . . . same body as before . . .
      end push

      procedure pop (var e : elementType)
          . . . same body as before . . .
      end pop

      depth := 0
  end stack

type smallStackOfString : instance stack (10, string)
type bigStackOfInt : instance stack (100, int)

var stringStack :  smallStackOfString
var intStack :  bigStackOfInt

stringStack.push ("Hi there")
stringStack.push ("Hello yourself")
```

Figure 5. A Polymorphic Stack Object Type in the Objective Turing Dialect and Example Instances.

class polymorphism to either static parametric polymorphism (using C preprocessor macros), or opaque polymorphism using pointers [1].

Objective Turing follows this same static parametric model and uses type parameters to provide polymorphic objects. Syntactically, we introduce a new declaration for a *type class,* which is simply a type-parameterized type declaration. The dialect allows type classes of any type, in particular object type classes.

Figure 5 shows the *stack* object type extended to be an object type class for stack object types with any depth limit and any type of elements. Instances of the object type class yield an object type, as shown in the declaration of the *smallStackOfString* type and its subsequent uses.

We can specify the syntax of type classes in TXL by adding the *typeClassDeclaration* as an alternative declaration form, *instanceType* as an alternative type specification form, and using the new productions:

```
define typeClassDeclaration
  order
      type class [id]  ( [list id] ) : [typeSpec]
end define

define instanceType
  order
      instance [id]  ( [list expnOrTypeSpec] )
end define

define expnOrTypeSpec
  choose
      [expn]
      [typeSpec]
end define
```

This grammar allows type classes to be parameterized by either a type or an expression. We will assume that expression parameters to type classes are to be passed by name, although pass by value could easily be implemented by the transform if required.

The basic strategy for transforming type classes is to delete the type class declaration itself and transform instances of the it into copies of the class' type specification with the actual argument types and expressions of the instance substituted for the formal parameters of the class. The main rule starts things off, applying the *fixTypeClasses* rule to every scope in the program before applying the *fixObjectTypes* rule.

```
rule mainRule
      replace [declarationsAndStatements]
      P [declarationsAndStatements]
  by [declarationsAndStatements]
      P [fixTypeClasses]
        [fixObjectTypes]
end rule
```

FixTypeClasses deletes the type class declaration and applies the *fixInstantiations* rule to search its scope of declaration for instances to transform.

```
rule fixTypeClasses
      replace [declarationsAndStatements]
      type class TCname [id] ( Formals [list id] ) :
          TCbody  [typeSpec]
      RestOfScope [declarationsAndStatements]
  by [declarationsAndStatements]
      RestOfScope
          [fixInstantiations TCname Formals TCbody]
end rule
```

FixInstantiations searches the scope for instances of the type class and replaces the instance clause with a copy of the type class' type specification body in which the actual arguments of the instance have been substituted for the formal parameters of the type class.

```
rule fixInstantiations TCname [id] Formals [list id]
                                        TCbody [typeSpec]
    replace [declaration]
    type ITname [id] :
        instance TCname
                ( Actuals [list expnOrTypeSpec] )
    by [declaration]
    type ITname :
        TCbody  [substitute Formals Actuals]
end rule

rule substitute Old [id] New [expnOrTypeSpec]
  replace [id]
    Old
    by [expnOrTypeSpec]
    New
end rule
```

The application of the *substitute* rule takes advantage of another feature of TXL - its ability to automatically apply rules to corresponding elements of two lists of items. In this case, *Formals* and *Actuals* are lists of *ids* and *expnOrTypeSpecs* respectively. TXL automatically applies the rule successively to each corresponding (*id, expnOrTypeSpec*) pair in the lists.

Figure 6 shows the result of using these transforms on the example polymorphic stack object type of figure 5, to yield monomorphic object types that can then be transformed by the *fixObjectTypes* rule to yield a true Turing language program.

Conclusion

We have shown that it is possible to clearly specify both object types and parametric polymorphism using independent syntactic transformation rules. These transformations have been expressed in the TXL dialect specification language and fed to the TXL processor to yield a viable prototype implementation of Objective Turing. With little significant change, these same transformation rules can be used for any compiled language of the Pascal family and so form a convenient and compact specification of the general features.

Two other features commonly associated with object-oriented programming are *inheritance* and *dynamic binding.* These have also been specified as independent TXL transformation rules and are features of the full Objective Turing dialect [10].

```
type smallStackOfString :
  object
    import (error)
    export (push, pop)

    var storage : array 1 .. maxdepth of string
    var depth : 0 .. 10

    procedure push (e : string)
        . . . same body as before . . .

    end push

    procedure pop (var e : string)
        . . . same body as before . . .

    end pop

    depth := 0
  end smallStackOfString

type bigStackOfInt :
  object
    import (error)
    export (push, pop)

    var storage : array 1 .. maxdepth of int
    var depth : 0 .. 100

    procedure push (e : int)
        . . . same body as before . . .

    end push

    procedure pop (var e : int)
        . . . same body as before . . .

    end pop

    depth := 0
  end smallStackOfString

var stringStack :  smallStackOfString
var intStack :  bigStackOfInt

stringStack.push ("Hi there")
stringStack.push ("Hello yourself")
```

Figure 6. Result of Applying the *fixTypeClasses* Rules to the Example of Figure 5.

Scope and Limitations of TXL

The TXL transformations given in this paper, while quite viable, are deceptively simple and transparent, in part because of the necessity of simplifying the syntax of the dialect for concise presentation in this paper. In practice, rule sets can be much larger than the set given here, and it is often very difficult to see how to achieve the desired result using TXL transformation rules.

The scope of possible transformations implementable in TXL is however much greater than the simple syntactic transforms shown in this paper. TXL rule sets are capable of implementing all of the traditional programming language implementation tasks, including parsing, name and scope analysis, type checking, operator precedence analysis, interface matching, anti-aliasing, and even (in theory) code generation. In fact, it has been shown that the computational power of TXL rule sets is equivalent to that of Turing machines [11], so there is no theoretical limit to the translation that can be done short of the bounds of computability.

From a practical standpoint, however, TXL is severely limited by the complexity of the rule sets and by the performance of the TXL processor, which necessarily uses a variant of Prolog unification to apply rules. The performance of the rule set given in this paper is quite reasonable, taking about 8 seconds to translate a 200 line Objective Turing program into Turing on a Sun 3/50, but it is very easy to write rule sets that can take much longer. A detailed discussion of the scope and limitations of TXL appears in [11].

Acknowledgements

TXL was designed by J.R. Cordy and C.D. Halpern at the University of Toronto, and was implemented by the authors at Queen's University. The Turing programming language was designed by R.C. Holt and J.R. Cordy at the University of Toronto. The ideas for the run-time models used in the transformations described in this paper come from various sources, including the implementations of C++ [1], Euclid [7] [8] and Force One [9]. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. B. Stroustrup, *The C++ Reference Manual,* Addison-Wesley, 1986.

2. B.J. Cox, *Object Oriented Programming : An Evolutionary Approach,* Addison-Wesley, 1986.

3. R.C. Holt and J.R. Cordy, "The TURING Programming Language", Comm. of the ACM 31,12 (December 1988), pp. 1410-1423.

4. J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", Proc. IEEE 1988 International Conference On Computer Languages, October 1988, pp. 280-285.

5. R.D.Tennent, *Principles of Programming Languages,* Prentice-Hall, 1981.

6. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys 17,4 (December 1985), pp. 471-522.

7. B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek, "Report on the Programming Language EUCLID", ACM SIGPLAN Notices 12,2 (February 1977).

8. R.C. Holt and D.B. Wortman, "A Model for Implementing Euclid Modules and Prototypes", ACM Trans. on Programming Languages and Systems 4,4 (October 1982), pp. 552-562.

9. G.V. Cormack and A.K. Wright, "Polymorphism in the Compiled Language Force One", Proc. HICSS-22 1987 Hawaii International Conference on System Sciences, Volume II (Software), January 1987, pp. 284-292.

10. J.R. Cordy and E. Promislow, “Specification and Automatic Prototype Implementation of Object-Oriented Concepts Using the TXL Dialect Processor”, External Technical Report 89-251, Department of Computing and Information Science, Queen’s University at Kingston, March 1989.

11. Eric Promislow, A Run-Time Model for Generating Semantic Transformations from Syntactic Specifications, M.Sc. thesis, Department of Computing and Information Science, Queen’s University at Kingston, September 1989.