SOFTWARE TEMPLATES

R. B. Kieburtz
&
R. G. Babb

OGC TR CS/E 83-006

Oregon Graduate Center
Dept. of Computer Science & Engineering
19600 N.W. Walker Road
Beaverton, OR 97006

(503) 645-1121

Abstract

Software templates are high-level specifications of algorithms, given in a notation that is intelligible to a human reader, and independent of any particular computing environment. The intent is that such a specification should be directly translated into a computer program, when it is called out of a library and bound to a particular computing environment. The research proposed here will attempt to determine the feasibility of automating this translation process, by drawing upon recent results in formal semantic specification, program transformation, and semantics-directed compilation.

SOFTWARE TEMPLATES

Despite numerous attempts to make programming easier by designing better languages, working out logics for program verification, developing concepts of program modularity, and improving the tools used to construct programs, programming remains a difficult, demanding, and labor-intensive activity. To develop the ability to re-use validated program segments as components of programmed systems has long been a goal of programming methodology. This goal has proven to be an elusive one, not only when a new system is designed, but especially in the maintenance of existing systems. It motivates the research proposed here.

The reason that programs are so difficult to use as components of systems other than the ones for which they were specifically designed, is that each program is a concrete object, configured to fit exactly into the mold in which it is cast.

Consider, for instance, the common task of reducing strings of blanks to single occurrences, in a stream of characters. When expressed in Pascal, this program is tailored to an environment expressed by the Pascal types used to construct the representation of a stream of characters. If the program removes blanks from a file of characters, for instance, it doesn't bear much resemblance to a program that removes blanks from an array of characters. The problem is not restricted to strongly-typed languages; exactly the same specialization to its environment would be found in a program coded in "C" or in an assembler language.

How, then, are we ever to achieve interchangeability of programs as components of systems? The Ada[1] approach is to provide a standard language as a first step towards standardizing the environments in which program components are to be embedded. This is analogous to the standardization that makes it possible to build electronics systems from components. If you stick to a common technology, then all the components use common dc voltages, and have compatible power requirements and switching times. Programs are not electronics components, however, and will not easily be forced into a "common technology" mold.

We believe the answer will be found at the level of abstract program specifications, rather than by viewing concrete programs as interchangeable components. The correct analogy with electronics is not the off-the-shelf hardware component, but a design in the VLSI cell library, or better yet, the high-level functional

[1]Ada is a trademark of the U.S. Department of Defense.

specification from which the cell layout was derived.

In this research we propose to address the problem of reusable program components from the point of view of abstract specifications. These specifications must account not only for the functionality of programs, but also for the environments in which the programs can work. Concrete programs are to be derived from abstract specifications; this is a process that we believe can be largely automated. We call these reusable specifications *software templates*.

1. Template Specifications

Our notion of template specifications is based upon the theory of abstract data types that has been developed over the past ten years [ADJ77, ADJ78, Gut77, GuH78, LiZ74, Par72]. This style of specification is precise, has a well-defined abstract semantics, is constructive (in the sense that algorithms are specified), and provides for the use of generic typing [Mil78], allowing general-purpose templates to be specified. Formal verification of abstract specifications is feasible, although difficult, whereas for most concrete programs it is infeasible because of the complexity of the programming language semantics. Efforts to achieve correctness should be concentrated at the specification level.

A template specification is completely independent of implementation environment and language. It does not unduly constrain the choice of data representations, yet it contains all the information necessary to derive a program component and to match it to its environment. What we hope to demonstrate is that good (i.e. efficient) programs can be automatically derived from abstract, equational specifications, and that human interaction can be effectively employed to aid in the process.

1.1. Deriving program components from templates

An abstract data type is an algebra [ADJ78] whose functions are specified by a set of defining equations, and whose carrier set is unspecified. There are two more or less separable parts to the derivation of a program from an abstract specification, (i) choosing a representation, and (ii) translating equations into program code. The questions of automating both of these aspects of programming have been studied separately, [Dew79, SSS81, HOD82], but both problems need further work in order to make their automation practical.

We don't propose to tackle the enormously difficult problem of developing an algorithm to choose appropriate data representations automatically. When a software template is instantiated relative to a particular environment, much of the choice of

representation will be constrained by the environment, and by the representation choices that have been made for other components of the same program. That choice which remains is best exercised by a human programmer, using an automated template instantiation tool. Initially, a programmer may have to spend a lot of effort in choosing appropriate representations for a new class of environments, such as that provided by a new operating system, or a new programming language. Once chosen, however, the representations of commonly used abstract data types will be entered into a catalog of representations, and used by a template instantiator without further troubling the human programmer.

We advance the hypothesis that translation of equations into efficient program code can be made feasible with a reasonable amount of effort, and we intend to demonstrate a system that will accomplish this. In order to make translation of equations possible, one must know what the equations mean, in a very strict sense. Our notion of the meaning of equations is based upon a mathematical semantics of equations that define functions [Kie82].

A set of defining equations will have well-defined semantics if the following constraints are satisfied [ODo77]:

i) No function variable may be repeated on the left-hand side of an equation;

ii) If the left-hand sides of two equations are unifiable, then the right-hand sides must be the same after application of the unifying substitution;

iii) It must not be possible for a well-formed term to contain an (overlapping) critical pair of redices.

Constraints i), ii), iii) are sufficient to ensure that if a set of equations is directed as term-rewriting rules, then for any term which has a reduction to a normal form, that normal form will be unique. This property is required if a set of equations is to define functions.

The constraints i), ii), iii) are not enough to guarantee that a set of defining equations constitutes a complete specification, however. A potential hazard is that the equations do not specify the behavior of a function on all objects of the abstract data type. In order to avoid this difficulty, we require that the objects of a data type must all be representable by a *basis* of terms, freely generated by a finite set of generators. This requirement is not confining; it merely restricts us to enumerable domains of objects. A further constraint that we impose upon defining equations is

iv) The terms appearing as argument expressions in the left-hand sides of definitions must be basis terms.

This constraint subsumes i) and iii), and is also not altogether necessary, but together with ii) it is sufficient to ensure

decidability of whether or not two left-hand sides overlap, and of whether or not a set of equations defines the behavior of a function on all canonical terms. Constraint iv) obviates the need to postulate "sufficient completeness" [GuH78], a property which is not decidable, in general, unless constraint iv) is satisfied.

When a set of equations has a semantics which gives it meaning as a set of computable functions, then the translation of equations to recursive function definitions is not difficult. So in one sense, the translation of specifications to programs is an easy problem [HOD82]. However, we wish to obtain translations into programs that are not necessarily to be given in an applicative programming language. For this purpose, we need to address the transformation of recursively-defined functions into iterative schemes, whenever this is possible, and the generation of code in a (possibly imperative) programming language not of our own choosing. In this context, we mean by a programming language any notation that can be used to express commands, or demands for evaluation, to a computer system. This includes sequences of calls upon system-defined primitives, an operating system's command language, or database commands as well as the familiar compiled programming languages.

Transformation of recursion schemes to iteration schemes is a problem in which we, along with others, have invested considerable effort [BuD77, DaB76, Fea82, KiS81]. We propose to continue the implementation phase of that effort as part of the experimental research we propose here.

Note that our approach differs from the approach of ***transformational implementation*** [BGW77, BrP81, Wil81] in a significant detail. We do not propose to put the human programmer into the decision process in which transformations are selected. Our experience with transformations makes us pessimistic that a human will be able to follow the steps of successive transformation of a program's form. We advocate using the judgement of the human programmer in the selection of algorithms (when specifications are written) and of data representations. In some sense, our plan intends a realization of the concepts of "putting theories together" [BuG77].

Generation of programs from iterative, functional program schemes appears not to be too difficult, if it is assumed that choice of representation will be taken care of by an interactive human user whenever it cannot be done by directly instantiating representations drawn from a library. The problem of program generation we contemplate is simpler than the more general problem of semantics-directed compilation because our defining equations have a direct semantics.

When a program is assembled out of a number of software templates, the interfaces between templates are specified in abstract terms, but are not yet cast in concrete. This affords an opportunity for considerable optimization of the resulting program. Most of this optimization can be based upon well-known techniques of data-flow analysis, and will result in economies such as copy elimination, avoiding redundant computation, and avoiding the evaluation of unreferenced expressions. These are the measures needed to derive programs from abstract specifications that will run as economically as those carefully coded in an imperative programming language.

2. An example: Removing excess blank characters from text

As an example of what we have in mind, let's consider a software template to meet a very simple requirement: reducing each string of blank characters in a stream of text to a single blank character. We begin by specifying the data type of a blank remover

$squeezeblanks : seq(\alpha) \rightarrow seq(\alpha)$

where α is a type variable. The specification of *squeezeblanks* can make use of any of the functions defined for the type $seq(\alpha)$. These are:

Generators:

$nilseq : seq(\alpha)$

$apndr : seq(\alpha), \alpha \rightarrow seq(\alpha)$

Extractors:

$first : seq(\alpha) \rightarrow \alpha$

$rest : seq(\alpha) \rightarrow seq(\alpha)$

where

$first(apndr(nilseq, x)) = x$

$first(apndr(apndr(s, y), x)) = first(apndr(s, y))$

$rest(apndr(nilseq, x)) = nilseq$

$rest(apndr(apndr(s, y), x)) = apndr(rest(apndr(s, y)), x)$

In order to write the equations specifying *squeezeblanks* we find that α cannot stand for just any type, but for a type which defines an operation

$isblank? : \alpha \rightarrow boolean$

Equations defining *squeezeblanks* are:

$squeezeblanks(nilseq) = nilseq$

$squeezeblanks(s) =$
if $isblank?(first(s))$
then $apndl(first(s), squeezeblanks(stripblanks(rest(s))))$
else $apndl(first(s), squeezeblanks(rest(s)))$
fi
where $s = apndr(w, x)$
andwhere $apndl(x, nilseq) = apndr(nilseq, x)$
$apndl(x, apndr(w, y)) = apndr(apndl(x, w), y)$

The function *stripblanks* is necessary for the definition of *squeezeblanks*, but is not by itself a part of the abstraction we are trying to capture. If the abstraction were packaged as an abstract data type, *stripblanks* would not be mentioned among its externally visible operators. It is defined by:

$stripblanks(nilseq) = nilseq$

$stripblanks(s) =$ **if** $isblank?(first(s))$
then $stripblanks(rest(s))$
else s
fi
where $s = apndr(w, x)$

In giving the specification of *squeezeblanks* it was necessary to define an additional function on the type $seq(\alpha)$. Notice that it is not necessary that this function, *apndl*, actually be defined in the specification of $seq(\alpha)$; it is an *auxiliary* function needed to describe the algorithm which defines *squeezeblanks*. The function *apndl* may or may not have any apparent realization in a program for *squeezeblanks*.

The first step in deriving a program from a specification is always the same, independent of the environment. The equational specifications are interpreted to give a recursive function definition. Because of the constraints (ii) and (iv), the separate equations defining a function lead directly to separate arms of a conditional whose predicate(s) can be inferred from the equations. In the case of our example, the function definition becomes:

```
squeezeblanks = λs. if isnilseq? (s) then nilseq
                    else apndl (first (s),
                                squeezeblanks
                                  (if isblank? (first (s))
                                   then stripblanks (rest (s))
                                   else (rest (s))
                                   fi))
```

In the right-hand side of this "program", the predicate *isnilseq*? distinguishes (the representation of) a sequence whose value is *nilseq* from one which is not.

The second step is the transformation of a recursive defining form into an iterative form, whenever this is possible. The defining form is first rewritten as a combinator expression[2] by eliminating variables. The combinator form of a recursive function definition will then be examined by a program transformation system to determine whether it can be replaced by a semantically equivalent iterative form. We have studied and cataloged a class of transformations on program schemes that we believe to be generally useful [KSh81]. We are currently developing a scheme recognition system [KiG82] that will be capable of recognizing programs for which a cataloged transformation is applicable, and of applying the transformation.

In the case of our example, such a transformation is possible. The functions *apndl* and *apndr* satisfy the condition of being *associative duals* with pivot *nilseq*, allowing the definition of *squeezeblanks* to be subjected to the transformation given as Proposition 2 of [KSh81]. The equivalent form is[3]:

```
squeezeblanks = 1st∘(while not∘isnilseq?∘2nd
                           [apndr∘[1st, first∘2nd],
                            if isblank?∘first∘2nd
                            then stripblanks∘rest∘2nd
                            else rest∘2nd
                            fi]
                      ∘[nilseq, id]
```

Notice that in this form, the auxiliary function *apndl* which had been defined in order to give the equational specification of

---

[2]Instead of the primitive combinators S, K, and I of combinatory logic, we find the combining forms of FP [Bac78] to be more convenient.

[3]The transformed function representations are given in slightly modified FP notation. The symbol ∘ represents functional composition, $[f,g]$ is the construction of a functional pair, 1^{st} and 2^{nd} are selectors on functional pairs, the conditional is written as **if**...**then**...**else**...**fi**, and the form (**while** γ ρ) is equivalent to **if**

squeezeblanks has disappeared. It does not require any concrete representation.

The function *stripblanks* can also be transformed to an iterative form. If we have incorporated into our program transformation system the result $isnilseq?(s) = true \Rightarrow nilseq = \mathbf{id}(s)$ then the definition of *stripblanks* can be written in FP notation as:

$$\begin{aligned} stripblanks = {} & \mathbf{if}\ or \circ [isnilseq?, not \circ isblank? \circ first] \\ & \mathbf{then\ id} \\ & \mathbf{else}\ stripblanks \circ rest \\ & \mathbf{fi} \end{aligned}$$

This is an instance of a particularly simple form of recursion scheme which has as an equivalent iterative form

$$stripblanks = (\mathbf{while}\ \ and \circ [not \circ isnilseq?, isblank? \circ first]\ \ rest)$$

The functional program for *squeezeblanks* is obtained by substitution of the iterative form for *stripblanks* into the iterative form previously obtained.

What remains is to define representations for the functions *nilseq*, *apndr*, *first*, *rest*, *isnilseq?* of the abstract data type $seq(\alpha)$, and for the combining forms $(\mathbf{while}\ \gamma\ \rho)$ and **if...then...else...fi** in the environments in which *squeezeblanks* is to be realized. If these environments are programmable, then it is highly likely that the combining forms will be closely realized by primitives of the programming language. The construction combining form, "[...]" and the selector functions 1^{st} and 2^{nd} of FP will have to be realized by the use of variables, in conventional, programmable environments.

The task that requires (for the present, at least) human intervention is the identification of abstract functions with functions provided by the environment in which instantiation of a software template is to occur.

2.1. Environment 1: Pascal files

As our first example, let's apply the redundant blank removal template to a file of characters, as provided by Pascal. The programmer must come up with a list of identifications of abstract functions with program forms available in the environment. Since this environment is an imperative programming language, in which some operations produce side effects on the environment, it will not be possible to identify each abstract operator, which is a function, with a single concrete operator of the environment. Instead, we shall identify *combinations* of abstract operators with *sequential combinations* of concrete operators. In this example, our

---

γ **then (while** $\gamma\ \rho$**)**$\circ\rho$ **else id fi**.

Pascal expert makes the following identification:

seq(α)	:	**file of** char
isnilseq? s	:	eof(s)
nilseq	:	rewrite(<file variable>)
(*first*, *rest*) s	:	(s↑, s) **after initial** reset(s)
	:	(s↑, s) **after** get(s)
apndr (s,x)	:	s **after** s↑ := x; put(s)
isblank? x	:	x = BLANK

The ambiguity in the identification of the pair (*first*, *rest*) reflects the fact that the Pascal environment is characterized by a state which is not part of the abstraction. In this case the ambiguity is resolved by use of the **initial** specification; we make no claim that this mechanism is adequate to resolve all state specification ambiguities.

The combining forms **while** and **if...then...else...fi** are readily identified with Pascal control structures. This identification is actually part of the task of environment specification and is a job for the human programmer. The realization of *squeezeblanks* which results from the identification is:

```
var   infile, outfile : file of char;
      x : char;
begin
   reset(infile);
   rewrite(outfile);
   while not eof(infile) do
      begin
         x := infile↑;
         outfile↑ := infile↑;
         put(outfile);
         get(infile);
         if x = BLANK then
            (* stripblanks *)
            while not eof(infile) and (infile↑ = BLANK) do
               get(infile)
      end
end
```

This realization may be incorporated into a Pascal program either as a macro (assuming that provision is made to avoid clashes of identifier names, and to insert declarations at the appropriate places) or as a closed procedure.

There are a couple of interesting aspects to the translation from a functional program specification into Pascal. One is that since applications of the functions *first*, *rest* are implemented in Pascal by a side effect of the procedure **get**, multiple occurrences

of an expression involving an application of *first* or *rest* must be collapsed into a single call upon the procedure **get**. In order to accomplish this, it has been necessary to introduce a variable, x, to save the state of the file buffer of infile.

A second observation is that we have been using a functional notation in which certain operators (the sequence constructor *apndr* and the construction combining form) have non-strict (i.e. "lazy") semantics. In consequence, it is possible to give non-strict interpretations to some functions that act upon these forms, such as "and", which applies to a two-element construction. A non-strict "and" is defined as if curried,

and(*false*) = **false**

and(*true*) = **id**

The first of these two applications of "and" results in a constant-valued function, the second does not. When a non-strict "and" is translated into Pascal, one has to watch out for the fact that the semantics of the corresponding Pascal operator is defined neither as strict nor as non-strict (the ISO standards committee copped out on this issue) and different implementations of Pascal can and do differ. We have assumed a non-strict Pascal "and" in the example above.

2.2. Environment 2: Standard files in 'C'

Changing the example slightly, suppose the environment provided is a 'C' program. Our 'C' expert might provide the following identifications:

seq(α)	:	#include "stdio" **char** ch;
isnilseq? s	:	ch == NUL
nilseq	:	{null expression--standard output is preinitialized}
(*first*, *rest*) s:		(ch, stdin) **after** getc(ch)
apndr (s,x)	:	stdout **after** putc(x)
isblank x	:	x == BLANK

In this example, there is no ambiguity about which concrete operation to use, depending upon the state of the system. However, there is a need to associate a particular, declared variable with an input file, as the function *isnilseq*? is realized not by a test upon a file variable in 'C', but by a test upon a value read from the file. Thus our programmer has bound a program variable "ch", by giving its declaration in the identification list. The 'C' code resulting from this identification will be:

```
#include "stdio"
char ch;
while (getc(ch) ~= NUL)
   {putc(ch);
    if (ch == BLANK)
      {while ((getc(ch) ~= NUL) && (ch == BLANK))
         ; }
   }
```

2.3. Environment 3: A Pascal array representation

For another application, we might wish to use a representation of a string as an array of characters. Our Pascal wizard makes the following identification:

```
seq(α)          :  record
                      str : packed array[1..maxinx] of char;
                      inx : 1..maxinx+1; (* with succ *)
                      length : 0..maxinx; (* with succ *)
                   end
                   where maxinx = 2048
isnilseq? s     :  s.inx > s.length
nilseq          :  s after s.length := 0; s.inx := 1
(first, rest) s:  (s.str[s.inx], s) after initial s.inx := 1
                :  (s.str[s.inx], s) after s.inx := succ(s.inx)
apndr (s,x)     :  s after  s.str[s.length] := x;
                            s.length := succ(s.length)
isblank?   x    :  x = BLANK
```

This representation differs from those of the first two examples in that it makes use of no environment-defined data type that is naturally suited to the abstraction to be realized. Substantial care is required by the programmer to ensure that the representation simulates the abstraction. This may be a fruitful level on which to apply the techniques of program verification, since the individual code sequences are short, and the abstract specifications are given precisely. However, such verification is beyond the immediate scope of the presently proposed research.

The program derived from the identification is:

```
const MAXINX = 2048;
      MAXINX1 = 2049;
type  seq = record
                str : packed array [1..MAXINX] of char;
                inx : 1..MAXINX1;
                length : 0..MAXINX;
            end;
var   strin, strout : seq;
      lastch : char;
begin
   strin.inx := 1;
   strout.length := 0;
   strout.inx := 1;
   while not (strin.inx > strin.length) do
      begin
         strout.str[strout.length] := strin.str[strin.inx];
         strout.length := succ(strout.length);
         lastch := strin.str[strin.inx];
         strin.inx := succ(strin.inx);
         if lastch = BLANK
            then while not (strin.inx > strin.length)
                        and (strin.str[strin.inx] = BLANK) do
                        strin.inx := succ(strin.inx)
      end
end
```

This is an example of a program that is particularly tedious to code (or recode) by hand.

2.4. Alternate forms of specification

Equational specifications have some advantages. Among them are generality, familiarity, and to those who think readily in terms of functions, the ability to describe algorithms in terms of applicative expressions. Not everyone thinks of algorithms in this way, however, and for some classes of problems, a description of algorithms in terms of state transitions is most natural.

A task which acts on a single input stream to produce one or more output streams can be described as a *generalized sequential machine* (g.s.m.) [Gin62]. A g.s.m. is characterized by

i) a finite set of states, S;
ii) an input alphabet, Σ_I;
iii) a finite set of output alphabets, Σ_j;
iv) a state-transition function, $\sigma : S \times \Sigma_I \rightarrow S$;
v) a finite set of output functions, $\eta_j : S \times \Sigma_I \rightarrow \Sigma_j^*$
vi) a distinguished start state, $s_0 \in S$.

A g.s.m. has a formal characterization which can be automatically interpreted, if it is given in machine-readable form, and if the state-transition and output functions are specified in such a way that they can be interpreted. If each output function is restricted to produce only a bounded output sequence, then output function η_j can be constructively specified using conditionals, predicates over Σ_I, and finite sequences over Σ_j.

For example, the specification of *squeezeblanks* in this form is particularly simple. The input and output alphabets are identical; call them Σ, and suppose that Σ contains a distinguished element, *eos* which occurs only at the end of a sequence. There are three states, and the state transition function is:

$(s_0, eos) \longmapsto s_1$
$(s_0, \text{BLANK}) \longmapsto s_2$
$(s_0, x \in \Sigma - \{eos, \text{BLANK}\}) \longmapsto s_0$

$(s_2, eos) \longmapsto s_1$
$(s_2, \text{BLANK}) \longmapsto s_2$
$(s_2, x \in \Sigma - \{eos, \text{BLANK}\}) \longmapsto s_0$

The output function is:

$(s_0, \text{BLANK}) \longmapsto \text{BLANK}$
$(s_0, eos) \longmapsto eos$
$(s_0, x \in \Sigma - \{\text{BLANK}, eos\}) \longmapsto x$

$(s_2, \text{BLANK}) \longmapsto \varepsilon$
$(s_2, eos) \longmapsto eos$
$(s_2, x \in \Sigma - \{\text{BLANK}, eos\}) \longmapsto x$

The initial state is s_0.

Graphical descriptions of g.s.m.'s would be convenient in formulating specifications, but without some effort, they are not machine-readable.

In translating g.s.m.'s, sequence data types have special significance. In particular, any function applied to *eos* may have a special representation, as may an output of *eos*. Giving these representations is a task for the programmer. Returning to our example, suppose that a Pascal programmer has decided upon a

representation of sequences in terms of Pascal files. The representation is specified by:

input stream	:	**var** f : **file of** char; *init* reset(f)
output stream	:	**var** g : **file of** char; *init* rewrite(g)
input token = eos	:	eof(f)
output eos token	:	{no operation}
scan input token	:	get(f)
emit output x	:	g↑ := x; put(g)

The Pascal program segment produced from the g.s.m. specification and the chosen representation will be:

```
type states = (s0, s1, s2);
var  f, g : file of char;
     s : states;
begin
   s := s0; reset(f); rewrite(g);
   while not eof(f) do
      begin
         case s of
            s0 :  if eof(f) then s := s1
                  else if f↑ = BLANK then
                     begin s := s2; g↑ := f↑; put(g) end
                  else begin g↑ := f↑; put(g) end;
            s1 :  ; (* skip *)
            s2 :  if eof(f) then s := s1
                  else if f↑ = BLANK then (* skip *)
                  else  begin s := s0;
                           g↑ := f↑ := f↑; put(g)
                        end
         end;
         get(f)
      end
end
```

This example, constructed by hand, could be optimized by recognizing that the state s_1 is vacuous in the sense that no further output is produced once it is reached, and that the tests for *eos* within the cases for states s_0 and s_2 are unreachable and can be eliminated. However, the point we wish to make is that specifications given as g.s.m. mappings can be realized by very stereotyped program forms, of which the one above is a simple example.

Notice that in the last example, the program structure is radically different from that of the first example although both use the same Pascal environment, and both have chosen the same data representation for sequences. This is because the algorithm is determined by the abstract specification, and the two algorithms

differ.

3. Scope of the proposed research

We propose an experimental investigation of a system to instantiate software templates in specified environments. The examples shown in the preceding section illustrate the kind of capabilities that our system is to exhibit, and which we believe are achievable.

Our prior work has given us confidence that we can derive recursive function definitions from systems of defining equations, and can manipulate the intensional forms of functions to produce iterative forms suited to direct representation in a lower-level programming environment. We believe the most difficult step to automate, in instantiating a program template, is the choice of suitable representations for abstract objects, in terms of the types of objects provided by an environment. We intend to rely upon human interaction to accomplish this step.

The template instantiator must have considerable, although not exhaustive, knowledge about the programming environment in which a program component is being created. Ideally, this knowledge should be contained in an external (to the instantiator) specification of the environment. Externalizing this knowledge may indeed be possible. Initially, however, we shall probably embed some assumptions about a class of environments into our prototype instantiator. These assumptions will include knowledge about program variables, some rules about how atomic statements (or functions) are composed, knowledge of conditional and **while** forms, and knowledge of assignment or other binding mechanisms.

This is a project with ambitious goals, but we already have a foothold upon achieving them, by virtue of having constructed several subsystems. These include a tool [Rol82] which understands concrete and abstract syntax, and constructs a translator to abstract syntactic notation from a suitably annotated parsing grammar, a tool which interprets a system of equational specifications to give the normal form of any term, and a tool which computes the data types of functions defined by a set of equations, and which accommodates polymorphic types. We also have interpreters for several functional languages, including FP.

We are presently developing a tool that will apply cataloged program transformations, recognizing programs which are instances (in quite a general sense) of commonly occurring recursive program schemes. And we have available for study, an example of a contemporary experimental semantics-directed compilation system [JoS81]. This system uses externalized knowledge of a programming environment in order to synthesize programs for

that environment.

The research we propose here could be described as exploratory systems development. Its goal is to develop enough technology to show feasibility of the software templates approach to program design, and to uncover problems in use of the approach that we may not have forseen. Achieving high performance from an experimental prototype is not a goal. It is not intended that our prototype system will be suitable for use by others, except possibly as a research tool.

4. References

[ADJ77]
Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B., Initial algebra semantics and continuous algebras, **Jour ACM 24**, 1 (Jan. 1977), 68-95.

[ADJ78]
Goguen, J.A., Thatcher, J.W., and Wagner, E.G., An initial algebra approach to the specification, correctness, and implementation of abstract data types, in **Current Trends in Programming Methodology, Vol 4: Data Structuring** (R.T. Yeh, editor), Prentice-Hall, (1978), 80-149.

[Bac78]
Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, **Comm ACM 21**, 8 (Aug. 1978), 613-641.

[BGW77]
Balzer, R., Goldman, N. and Wile, D.S., On the transformational approach to programming, Proc. 2nd Inter. Conf. on Software Engr. (1976), IEEE, New York, 337-344.

[BrP81]
Broy, M. and Pepper, P., Program development as a formal activity, **IEEE Trans. on Software Engr. SE-7**, 1 (Jan. 1981), 14-22.

[BuD77]
Burstall, R.M. and Darlington, J., A transformation system for developing recursive programs, **Jour ACM 24**, 1 (Jan. 1977), 44-67.

[BuG77]
Burstall, R.M. and Goguen, J.A., Putting theories together to make specifications, Proc. 5th Inter. Joint Conf. on A.I., (1977), 1045-1058.

[DaB76]
Darlington, J. and Burstall, R.M., A system which automatically improves programs, **Acta Informatica 6**, 1 (Mar. 1976), 41-60.

[Dew79]
Dewar, R.B.K., *et al*, Programming by refinement, as exemplified by the SETL representation sublanguage, **ACM TOPLAS 1**, 1 (July 1979), 27-49.

[Gin62]
Ginsberg, S., Examples of abstract machines, **IEEE Trans. on Electronic Computers TEC-11**, 2 (1962), 132-135.

[GuH78]
Guttag, J.V. and Horning, J.J., The algebraic specification of abstract data types, **Acta Informatica 10**, (1978), 27-52.

[Gut77]
Guttag, J.V., Abstract data types and the development of data structures, **Comm ACM 20**, 6 (Jun. 1977), 396-404.

[HOD82]
Hoffman, C.M. and O'Donnell, M.J., Programming with equations, **ACM TOPLAS 4**, 1 (Jan. 1982), 83-112.

[JoS81]
Jones, N.D., and Schmidt, D.A., Compiler generation from denotational semantics, in **Lecture Notes in Computer Science 94**, Semantics Driven Compiler Generation, Springer-Verlag, 70-93.

[Kie82]
Kieburtz, R.B., Precise typing of abstract data type specifications, Proceedings of 1983 ACM Conf. on Princ. of Prog. Lang., (Jan. 1983).

[KiG82]
Kieburtz, R.B., and Givler, J., The recognition problem for functional program schemes, Research Memorandum (Aug. 1982), Dept. of Computer Sci. and Engr., Oregon Graduate Center.

[KSh81]
Kieburtz, R.B. and Shultis, J., Transformations of FP program schemes, Proc. of ACM Conf. on Functional Prog. Lang. and Computer Arch. (Oct. 1981), 41-48.

[Mil78]
Milner, R., A theory of type polymorphism in programming, **Jour. Comp. and Syst. Sci. 17**, (1978), 348-375.

[Moi82]
Moitra, A., Direct implementation of algebraic specification of abstract data types, **IEEE Trans. on Software Engr. SE-8**, 1 (Jan. 1982), 12-20.

[ODo77]
O'Donnell, M.J., Computing in systems described by equations, **Lect. Notes in Computer Science 58**, monograph (1977), Springer-Verlag, New York.

[LiZ74]
Liskov, B.H. and Zilles, S.N., Programming with abstract data types, **ACM Sigplan Notices 9** 4 (Proceedings of 1974 ACM Symposium on Very High Level Languages), 50-59.

[Par72]
Parnas, D.L., A technique for the specification of software modules with examples, **Comm ACM 15**, 330-336.

[Rol82]
Rollins, E.J., A syntax-analyzer constructor, Technical report CS/E-82-4, Oregon Graduate Center (1982).

[SSS81]
Schonberg, E., Schwartz, J.T., and Scharir, M., An automatic technique for selection of data representations for SETL programs, **ACM TOPLAS 3**, 2 (Apr. 1981), 126-143.

[Wil81]
Wile, D.S., Type transformations, **IEEE Trans. on Software Engr. SE-7**, 1 (Jan. 1981), 32-39.