Calculating Software Generators from Solution Specifications *

Richard B. Kieburtz   Francoise Bellegarde
Jef Bell   James Hook   Jeffrey Lewis   Dino Oliva
Tim Sheard   Lisa Walton   Tong Zhou

Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
PO Box 91000, Portland, OR 97291-1000 USA

October 14, 1994

Abstract

Software application generators can eliminate many of the technical aspects of programming for most computer users. We have developed a uniform approach to the design of program generators, based upon a simple idea—provide a declarative specification language for each application domain and give it a computable, denotational semantics. To make this idea practical, however, requires a comprehensive system for transforming and translating expressions in the higher-order functional operators of the semantics formulation into a reasonably efficient implementation expressed in a first-order, imperative programming language. This paper describes the system we have built to accomplish this.

The technique and the system have been applied to produce a generator for modules that validate and translate messages sent from a peripheral sensor to a central controller. The input to a generator is a specification of the data formats and data constraints that characterize a message. The output is an Ada package of six functions that perform message translation and validation.

*The research reported here has been sponsored by the USAF Materiel Command.

1 A technology for automatic program generation

Program generators offer a substantial reduction of effort to produce application-tailored versions of a common software design, but the task of designing and implementing a program generator for a new application domain can be formidable. This paper describes a new technology for creating program generators. It is built upon research results in the theory of programming languages, formal semantics, program transformation and compilation. It comprises a suite of translation and transformation tools that constitute a design automation system for software engineering.

In our method, the user's interface to a program generator is a language in which to specify each particular application for which a software module is required. We refer to this specification language as a *domain-specific design language* (DSDL), for it is tailored to the problem domain for which the generator is intended. A DSDL is a specialized, declarative language in which the important, high-level abstractions of the problem domain are directly expressible. Often, a DSDL is just a formalization of a tabular or graphical specification language that engineers in the problem domain have long been using to express detailed designs.

When a DSDL is used to provide input to a program generator, however, it must have been given a computational semantics. The requirements that we impose upon the semantics definition for a DSDL are that it be (i) compositional, (ii) effectively computable, and (iii) total. The implementation of a program generator is derived from the semantics of a DSDL through several steps of translation and transformation to obtain satisfactory algorithmic performance and to tailor the implementation to a specific platform and software environment.

Compositionality implies that an implementation can be assembled piecewise from the components of the semantics. Effective computablility eliminates reliance on the axiom of choice, for instance. The requirement that all semantic functions must be total allows us to use equational theories to drive program transformations.

The idea of deriving an implementation for a formally specified language from its semantics was first tried experimentally in the SIS system [20] over 25 years ago. However, at that time, the prospect of a technology to improve the performance of an implementation enough that it would become acceptable for practical use seemed remote. In the intervening years, there have been many discoveries relating to the formal calculation of programs, and it seems time to revisit the ambitious task of automating program generation.

2 Classes of transformations

The compositional style of programming used in designing a computational semantics for a formal specification language is attractive to the designer. However, powerful transformations are necessary to improve efficiency of the programs synthesized from the semantics. Semantics-preserving, fully automatic transformation tools relieve the software designer from having to consider programming details that tend to obscure high-level concepts relevant to the design itself.

The transformations we have considered fall into four classes, for which distinct implementation strategies seem most appropriate:

1. Parametric transformations are instances of general theorems established by parametricity arguments. They yield equivalences that apply in all datatypes, hence the resulting transformations are type-parametric.
2. Order-reduction transformations replace expressions that use higher-order functions by equivalent expressions using only first-order functions.
3. Algebra-specific transformations are those that depend upon some algebraic laws, such as the associativity and commutativity of a binary operator.
4. Architecture-specific transformations depend upon representation equivalences or operation code equivalences of a particular architecture. Such transformations are typically found in the code generator of an optimizing compiler.

A compositional style of programming introduces many intermediate data structures. Directly applying semantic functions may entail multiple traversals of the data structure that represents the abstract syntax of the language that is being interpreted. These problems can be addressed by two parametric transformation strategies:

- fusion or deforestation, in which identical control structures of sequentially applied functions are merged, often allowing an intermediate data structure to be eliminated [27, 9], and
- the tupling or parallel fusion strategy [6, 10]. in which a pair of functions that operate on the same data are transformed into a single function that returns paired results. Symbolically, this transformation is

$$(f\ x,\ g\ x) \Longrightarrow \langle f,\ g\rangle\ x$$

When applied to traditional functional programs, parametric strategies can require expensive and inexact analysis to determine whether sufficient conditions for their application are satisfied. However, if control structures are explicitly designated in the formulation of semantic functions and this information is preserved through the translation and transformation process, it can be exploited to drive the transformation strategies by pattern matching alone.

Parametric transformations are remarkably effective. However, they do not exploit specific, algebraic properties of functions used in designing a semantics. A property like the associativity and commutativity of multiplication over natural numbers is not parametric. Associativity is necessary to apply the accumulator-introduction strategy that eliminates recursion in favor of iteration. It can be exploited by transformation systems based on the unfold-fold method [10], but these require human intervention or *ad hoc* heuristics to direct them.

Term-rewriting, using a theory completion process for control, provides a flexible basis for implementing algebra-specific transformations [12, 4]. Such systems perform transformations on first-order programs. Parametric transformation strategies can also be performed by term-rewriting methods. Algebra-specific transformations are more costly and more difficult to automate than parametric transformations but they can have a dramatic impact on the performance of programs. Algorithmic complexity improvment can be obtained through transformations by a clever use of algebraic laws.

A strategy for order reduction is to generate a specialized version of each higher-order function for each distinct list of functional arguments to which it is applied in a given program. This specialization increases the size of the program but has no negative impact on its execution time. Generation of an appropriate data structure to represent closures [22, 1] leads to a more general but less straightforward approach for this class of transformations.

Specialization of functions can also be used to eliminate parametric polymorphism in a program. This may in turn allow an implementation to avoid boxed representations of data objects, which can be regarded as an architecture-specific transformation.

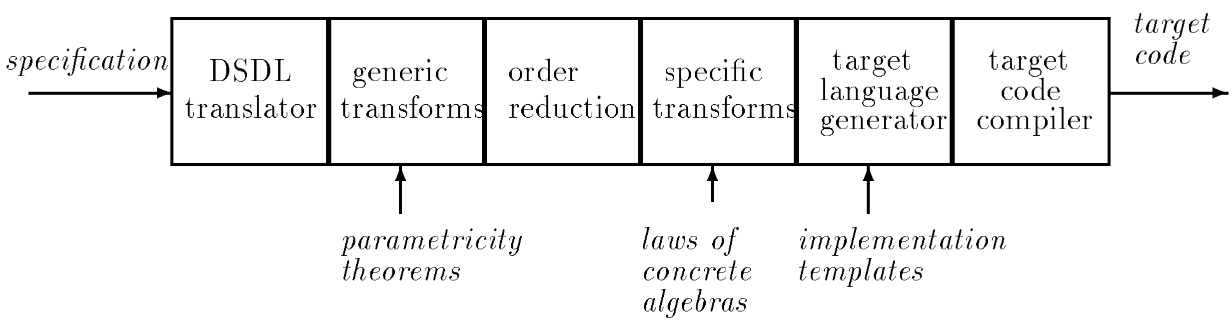

Figure 1—Transformation and translation pipeline

3 Computable denotational semantics

Denotational semantics for programming languages are translations of syntax to functional expressions such that all constructions are deterministic and composable. Composability implies that the semantics of a syntactic construction is a function of the semantics of its component parts—and of nothing else. If each of the semantic functions associated with a constructor of the abstract syntax is effectively computable, then we have a computable denotational semantics. Our tactic for making a specification language computable is to formalize its intuitive meaning in terms of a computable denotational semantics expressed in an executable meta-language.

We have designed the ADL language [15] as our preferred meta-language. ADL is an acronym for Algebraic Design Language. It adapts the notion of structure algebras from the mathematics of universal algebras to provide an unusually rich control structure without employing an explicit recursion operator. ADL is a language of total functions, which admits equational reasoning and program transformation by equational rewriting. ADL also incorporates a dual concept of coalgebras, which contribute control structures that correspond naturally to iteration.

3.1 Structure algebras in ADL

Some structure algebras, most notably the algebra of lists, are familiar to functional programmers and have been used by Bird, Meertens and their students [5, 17, 18, 13] to derive programs from logical specifications by formal reasoning. In ADL, structure algebras are first-class entities that can be declared, bound to identifiers, abstracted (in the module system) and form the basis for ADL control operators.

The declarative elements of ADL include *signatures* of algebraic varieties, *algebra specifications* and constant (value) declarations. ADL has program modules which are abstracted with respect to algebras.

Signature declarations do not use explicit recursion, for a signature does not define just a single algebra, but an entire class (or variety) of algebras that share a common structure. For example, the signature declaration (in abbreviated form) for *list* algebras is:

$$\textbf{signature}\ \mathit{list}(a)\,\{\textbf{type}\ c;\ \ \$\mathit{nil},\ \ \$\mathit{cons}\ \textbf{of}\ a * c\}$$

Each algebra in the variety defined by this signature has operators *$nil* and *$cons*. The identifier c, which ranges over all types, designates the *carrier* of an algebra of this variety. For each such algebra, c represents a specific type. The codomain of each operator is the carrier. The domain typing of each operator is specified in the signature. By convention, an operator symbol such as *$nil*, for which no domain typing is given, represents a constant of the carrier type. Signatures for multi-sorted algebras can also be written in ADL but shall not be discussed in this paper.

An algebra specification binds a type for the carrier and a compatibly typed constant for each operator symbol. For example, a *list*-algebra specification would be

$$\textbf{algebra}\ \mathit{Sum\_list} = \mathit{list}(\mathit{int})\{c := \mathit{int};\ \ \$\mathit{nil} := 0,\ \ \$\mathit{cons} := (+)\}$$

In this specification, both the type parameter, a, and the carrier have been bound to a common type, *int*; the operator symbol *$nil* has been bound to a constant of type *int* and *$cons* has been bound to the operator that designates *int*-addition.

Another *list*-algebra is a free term algebra, which has as its operators data constructors *nil* and *cons*, and which has as its carrier the set of terms constructed by well-typed applications of these operators. The type parameter, a, instantiated to any type, determines a particular instance of a free *list* algebra. Thus the carrier of a free term algebra derived from the variety *list* corresponds exactly to an instance of a *list* datatype in a functional programming language such as Standard ML [19]. For each variety declared by a signature in an ADL program, its free term algebra functor is declared implicitly.

In ADL, we distinguish two degrees of knowledge of the structure of an algebra. When an algebra is specified as an instance of a declared variety, we know how to form composite functions from it with the combinators described in the following section. This is what we mean by a structure algebra.

If the signature of the variety is not visible or the algebra has not been declared as an instance of a variety, then we know only its operators and their typings. We say that such an algebra is *concrete*. The definitions of operators of a concrete algebra may be invisible, if the algebra has been imported into a module. For example, the type *int* is the carrier of a concrete algebra which is externally specified. A concrete algebra is imported with its signature but without definitions of its operators. These definitions may even be implementation-defined.

To support reasoning about programs that depend upon concrete algebras, a concrete algebra may be partially axiomatized by a volume of equational laws. The laws are not part of ADL itself, but constitute an externally specified, logical constraint on the operators of a concrete algebra. When operators are implementation defined, it becomes an verification obligation to establish that the operators satisfy the required laws. The laws may then be used in formal verification or to justify program transformations. This convention supports the use of architecture-defined arithmetic in ADL.

3.2 Control structures in ADL

The expression elements of ADL include variables, constants, function and operator applications, datatype constructions, abstractions and saturated combinator expressions [1]. Of particular interest are the combinator expressions, for these determine all interesting control structures. ADL provides four combinators, *red*, *hom*, *gen* and *cohom*. The first two express control derived from algebras; the second two derive control from coalgebras. We shall only discuss here the algebraic control combinators.

The combinator *red* is indexed by a sort name and applied to an algebra specification. Its denotation is then a function from an initial term algebra to the carrier of the specified algebra. For example, the expression

$$red[list]\, Sum\_list \;:\; list(int) \rightarrow int$$

denotes a function that sums the elements of a list of integers. If this function were written in a language such as SML that has explicit recursion, you would recognize it as a recursion over the structure of its domain. However, the recursion is not explicit in ADL, it is instead calculated from the signature declaration given for the variety *list*. The combinator *red* is also called a catamorphism combinator [18].

The combinator *hom* generalizes the reduction scheme implicit in *red* to that of more general structure algebra morphisms. The domain of a *hom* expression need not be the carrier of an initial term algebra of the specified variety. Its domain may be any non-functional type. A *hom* is applied not only to an algebra specification, but also to a partition relation, which may be thought of as a map from the domain of a *hom*-expression into the carrier of a free term algebra of the required variety. A partition relation is specified with lambda-notation, as is a function, but the expressions it returns are applications of the operators of the signature of an algebraic variety.

For example, using the algebra specification given previously, we can supply a partition relation to obtain the function

$$hom[list]\, Sum\_list(\lambda n\ \mathbf{if}\ n\ \mathbf{mod}\ 2 \neq 0\ \mathbf{then}\ \$nil\ \mathbf{else}\ \$cons(1,\, n\ \mathbf{div}\ 2)) \;:\; int \rightarrow int$$

which calculates the integer part of the base 2 logarithm of a positive integer. The only semblance of *list* structure is in the sequential structure of the calculation, not in the data. However, the control structure of programs is exactly what we are interested in when looking for transformations to apply.

3.3 A tool for parametric transformations

A parametric transformation schema has an instance for every variety of structure algebra. The quintessential parametric transformation is based upon the Promotion Theorem [16]. This theorem and the transformation derived from it are most easily presented with the help of some notation from category theory.

The data of a signature with type parameter a consists of the domain typings of its operators. We can represent the structure of these data in the category $\mathcal{S}et$ by a coproduct of the domain types of the separate operators. This representation is the object map of a

[1] The term *combinator* is used here to mean an operator with no dependence on free identifiers and which operates on well-typed expressions in the language to produce a new expression. A combinator expression is *saturated* if all required arguments of the combinator are present.

bifunctor, $\mathcal{E}$. For instance, the bifunctor that represents the signature *list* has the object map

$$\mathcal{E}^{list}(a, c) = \mathbf{1} \; + \; a \times c$$

where $\mathbf{1}$ is the empty product. A *list*-algebra is represented in this notation by an arrow. For instance, the algebra *Sum_list* is the arrow

$$\mathcal{E}^{list}(int, int) \xrightarrow{\quad \{0, (+)\} \quad} int$$

where the curly brackets denote the **case** analysis of an element of a sum type, with component operators $0 : \mathbf{1} \rightarrow int$ and $(+) : int \times int \rightarrow int$.

The free *list* algebra with parameter type a is the arrow

$$\mathcal{E}^{list}(a, list(a)) \xrightarrow{\quad \mathbf{in}^{list} = \{nil,\ cons\} \quad} list(a)$$

where $\mathbf{in}^{list}$ is the composite operator of the free *list* algebra with carrier $list(a)$.

A *list* reduction, $h = red[list]\,\{c;\ f_{nil},\ f_{cons}\}$ satisfies the following set of equations

$$h\ nil \;=\; f_{nil} \tag{1}$$

$$h\,(cons(x, y)) \;=\; f_{cons}(x,\ h\ y) \tag{2}$$

that can be read from the commuting diagram:

$$\begin{array}{ccc}
\mathcal{E}^{list}(a, list(a)) & \xrightarrow{\quad \mathbf{in}^{list} \quad} & list(a) \\
{\scriptstyle \mathcal{E}^{list}(id_a, h)} \Big\downarrow & & \Big\downarrow {\scriptstyle h = red[list]\,\{c;\ f_{nil},\ f_{cons}\}} \\
\mathcal{E}^{list}(a, c) & \xrightarrow[\quad \{f_{nil},\ f_{cons}\} \quad]{} & c
\end{array}$$

Not only does $red[list]\,\{c;\ f_{nil},\ f_{cons}\}$ satisfy the equations read from the diagram, but it is the unique function for which the diagram commutes.

Moreover, for any variety T, every T-reduction is uniquely determined by a T-algebra specification and satisfies a similar diagram, in which the specific algebraic operators correspond to the T-signature.

Theorem: Promotion.
Let $\{c;\ f\}$ be a $T(a)$ algebra and let $g : c \rightarrow c'$. If there exists a $T(a)$ algebra $\{c';\ \phi\}$ such that $\phi \circ \mathcal{E}^T(id_a,\ g) = g \circ f$ with type $\mathcal{E}^T(a, c) \rightarrow c'$ then $g \circ red[T]\ f = red[T]\ \phi\ :\ T(a) \rightarrow c'$.

Proof: Consider the diagram below. The upper square commutes since h is a T-algebra reduction. The lower square commutes as the hypothesis of the theorem. Therefore the outer square commutes, thus the arrow on its right-hand edge is the unique T-algebra reduction determined by $\{c'\ \phi\}$.

$$
\begin{array}{ccc}
\mathcal{E}^T(a, T(a)) & \xrightarrow{\mathbf{in}^T} & T(a) \\
\Big\downarrow {\scriptstyle \mathcal{E}^T(id_a, h)} & & \Big\downarrow {\scriptstyle h = red[T]\, f} \\
\mathcal{E}^T(a, c) & \xrightarrow{f} & c \\
\Big\downarrow {\scriptstyle \mathcal{E}^T(id_a, g)} & & \Big\downarrow {\scriptstyle g} \\
\mathcal{E}^T(a, c') & \xrightarrow{\phi} & c'
\end{array}
$$

□

The higher-order transformation tool, HOT, uses a clever heuristic tactic to calculate an operator ϕ that satisfies the promotion theorem [24, 23]. The tactic is not complete—it does not always find a candidate if one exists—but it is inexpensive to apply and it often succeeds.

Given the data described in the proof of the Promotion Theorem, HOT introduces a symbol, g', with the assumed law that $g \circ g' = id_{c'}$. A consequence of the assumption is that $\mathcal{E}^T(id_a, g) \circ \mathcal{E}^T(id_a, g') = id_{\mathcal{E}(a,c')}$. Using this deduced law, we derive a representation for ϕ, namely that

$$\phi = g \circ f \circ \mathcal{E}^T(id, g')$$

Now g' is a meaningless symbol, but the expression on the right-hand side of the equation can often be simplified after introducing the detailed structure of f and of the bifunctor $\mathcal{E}^T$, which is derived from the signature T. In the course of simplification, any occurrence of the expression $g \circ g'$ is replaced by $id_{c'}$, which is justified by the assumed law. If, after simplification, the residual expression contains no occurence of the identifier g', then it represents the operator of a $T(a)$-algebra that was sought. Otherwise, the tactic fails.

4 Order-reduction transformations

Order-reduction transformations remove instances of higher-order functions (applications that include function-typed arguments or which return function-typed results) from a program while preserving its overall semantics. Obviously, this is only possible for programs that calculate ground-typed results from ground-typed data. The order-reduction stage in our translation pipeline consists of a suite of individual algorithms that perform specific order-reduction transformations efficiently. These are:

- A lambda-lifter [14], which removes nested function declarations and explicit abstractions, replacing them by new, closed function declarations and replacings occurrences of locally defined functions by applications of the new function constants. (Some nested declarations may be reintroduced during code generation, but if left in place throughout the pipeline, they might interfere with later transformation steps.) After lambda-lifting the program contains function definitions of the form $f\ x_1 \ldots x_n = e$ where each of the x_i is a variable and e is either a variable, a constant, an application, or a pattern case analysis.

- Eta-abstraction furnishes abstracted variables as arguments to an unsaturated application of a curried function. It is used to increase the arity of a function definition if its arity does not agree with its typing, and to supply additional, dummy arguments to an applicative expression that is unsaturated. When a function is polymorphically typed, like the polymorphic identity function, it may be applied with different arities. An instance of the polymorphic identity function applied to three arguments, for instance, can be replaced by a specialized identity defined by $id3\ x\ y\ z = x\ y\ z$.

  This transformation sometimes enables an expression in the body of a function declaration to be statically reduced, and is a prerequisite to further steps of function specialization and reduction. This transformation has been studied by Chin and Darlington [8], who refer to it as Algorithm A for higher-order function removal.
- Specializing a function to the arguments found at each of its call sites is a familiar technique for order-reduction (see for instance, Algorithm R of [8]). Specialization occurs in two phases. A naive but efficient algorithm is effective in nearly all cases that arise in practice. For cases that are beyond the scope of the naive algorithm, we have implemented a more general specializer based upon Reynolds algorithm [22].

For example, an application $map\ sqr\ x$, can be replaced by the application of a new function, $map\_sqr\ x$, whose definition is gotten by specializing the definition of map:

$$map\ f\ nil = nil \qquad map\ f\ x :: xs = (f\ x) :: (map\ xs)$$

with respect to the constant sqr, obtaining

$$map\_sqr\ nil = nil \qquad map\_sqr\ x :: xs = (sqr\ x) :: map\_sqr\ x$$

A sufficient condition for this technique to work is that the function-typed arguments in a definition are *variable or constant-only.* A function-typed argument of a higher-order function F is variable or constant only if in each recursive call in the declaration of F, this argument position is filled either by a variable or by a constant (i.e. a closed expression without free variables). The function map is variable-only. The definition $G\ f\ (a :: x) = (f\ a) :: (G\ (sqr)\ x)$ is constant-only. But the definition:

$$H\ f\ (a :: x)\ =\ (f\ a) :: (H\ (f\ \circ\ f)\ x)$$

is not variable or constant-only. Such arguments may cause a specializer to diverge as it attempts to specialize the function infinitely on arguments of growing size,

$$f, \quad f \circ f, \quad (f \circ f) \circ (f \circ f), \quad ((f \circ f) \circ (f \circ f)) \circ ((f \circ f) \circ (f \circ f)), \ldots$$

To specialize applications of higher-order functions that do not meet the restriction of using variable or constant-only arguments in the function's declaration, we have implemented Reynold's specialization algorithm. The method involves encoding as data the sequence of function-typed arguments generated by unfolding a recursive definition. The encoding is realized via a construction with a recursively-defined datatype. For simplicity, suppose that a function-typed argument f is transformed to $\mathcal{E}(f)$ in just a single recursive call of the higher-order function that is to be specialized. In the declaration of H above, $\mathcal{E}(f) = f \circ f$.

The first step in the transformation is to synthesize the recursive declaration of a datatype,

$$\textbf{datatype}\ `a\ T = C_0\ \textbf{of}\ `a\ \mid\ C_1\ \textbf{of}\ `a\ T$$

Next, replace the declaration of the higher-order function by that of a first-order function modeled on the same recursion scheme, except that

- when the function-typed argument, f, occurs in the first position in an application, it is replaced by $App_H\ f$;
- the composite function-typed expression $\mathcal{E}(f)$ that occurs as an argument in a recursive call of H is replaced by $C_1(f)$. For the example above, this new declaration is:

$$H'\ t\ x = App_H\ f\ x\ ::\ H'\ (C_1(f))\ x$$

The function App_H interprets the constructions in the recursive datatype. For the example, the following declaration is generated:

$$\begin{array}{rcl} App_H\ (C_0(f)) & = & f \\ App_H\ (C_1(t)) & = & (App_H\ t) \circ (App_H\ t) \end{array}$$

All that remains is to replace each application of the higher-order function in a program with a use of the new function applied to an appropriately constructed argument. That is, $H\ f$ is replaced by $H'\ (C_0(f))$.

This technique is more general than the naive specialization but it proliferates data types, so that a program becomes more difficult to analyze by subsequent transformation tools. We prefer to apply it only to the cases left by the naive specializer. Presently the order-reduction suite eliminates all occurrences of higher-order functions from programs except function-typed values embedded under data constructors. It is certainly possible to extend Reynold's technique to handle that case as well.

5 Algebra Specific Transformations

Many transformations are justified in part by the laws of specific algebras. In the ADL framework, an algebra that is imported as a parameter of a module is made concrete by declaring the operators of its signature as constants. However, nothing is revealed about the definitions of these operators, thus only their typings are visible in the scope of the module. We refer to an algebra imported in this way as "concrete". As a logical extension to the module, selected properties of a concrete algebra may be asserted as equational laws. It is these laws on which we base algebra-specific transformations. At the present time, there is no formal verification that the realization of a concrete algebra actually obeys the asserted laws. This gap in verifiability obviously needs attention in the future development of our system.

Commonly used equational laws are those of associativity, commutativity, distributivity, right and left unit, and right and left inverse. Laws justify tactics such as recursion elimination, which can sometimes reduce the asymptotic complexity of an algorithm. Unfortunately, it is very difficult to fully automate the application of such tactics.

Astre is a transformation tool based on rewriting techniques [2]. It is flexible enough so that some tactics can be fully automated. An example is the elimination of structural recursion by accumulator introduction in the presence of an associative operator, which is the familiar *foldr*-to-*foldl* transformation when specialized to *list* algebras.

A rewrite system is a set of rules, ordered pairs of terms, written as $l \rightarrow r$. When a first-order functional program is expressed by a set of mutually recursive pattern-matching equations, it translates into a rewrite system $R0$. The techniques that are used to transform such a program are simply rewriting and critical pair computation. A critical pair is the result of an *overlap* between the left-hand sides of two rules $g \rightarrow d$ and $l \rightarrow r$. An overlap exists if there is a position ω in l such that $l|_\omega$ and g are unifiable with the most general unifier σ (after renaming the two rules so that their respective sets of variables are disjoint). A *critical pair* is the (new) equation $\sigma(l[\omega \leftarrow \sigma(d)]) = \sigma(r)$ where the notation $t[\omega \leftarrow u]$ denotes the replacement in t of the subterm at position ω by u. Rewriting enables both folding and unfolding of definitions, depending upon the orientation of the equations as rules. Critical pair computation performs both instantiation and unfolding, hence providing an implemention for transformation by the unfold/fold methodology. This technique has been called *synthesis by completion* [11, 12].

In Astre, synthesis by completion is used as a mechanism to transform R_0 into a sequence of rewrite systems $R_1, R_2, \ldots, R_n$ to get from a functional program P_0 to a new, semantically equivalent program P_n that is more efficient. Astre translates R_n into an SML program where functions are presented by a set of mutually recursive functions with pattern-matching arguments.

A fully automated transformation system needs additional techniques, including

- a fully automated mechanism to discover rules that introduce new function definitions to form *synthesis rules* in the system. Critical pair computations with synthesis rules are the basis of many transformations. Synthesis rules were called *eureka* rules in the fold/unfold methodology because they depended upon the insight of a clever user.
- a mechanism to orient critical pairs into rewrite rules and to control critical pair production so that it generates a complete definition of the synthesized function. Astre controls the orientation of critical pairs into rules as required by the transformation strategy. It guarantees that termination of the rewrite system is preserved during the synthesis. Astre also carefully controls the production of critical pairs hence ensuring termination of the completion [3].

Consider, for example, the function that reverses the elements of a list. It is translated into the following rewrite system:

$$reverse(nil) \rightarrow nil \tag{3}$$
$$reverse(x :: xs) \rightarrow reverse(xs) \mathbin{@} [x] \tag{4}$$

where @ is a concrete algebra operator that is associative and has *nil* as right and left unit. A simple analysis discovers that the recursive call $reverse(xs)$ in the right-hand side of Rule 4 occurs under the associative operator @. In this case, it introduces automatically a synthesis rule $reverse(x) \mathbin{@} u \rightarrow g(x, u)$. This synthesis rule reduces the right-hand side of rule 4: $reverse(x :: xs) \rightarrow g(xs, [x])$. Critical pair computation with the right unit law, $x \mathbin{@} nil \rightarrow x$, gives the pair $(reverse(x), g(x, nil))$, which yields a new definition of *reverse*:

$$reverse(x) \rightarrow g(x, nil) \tag{$*$}$$

Critical pair computation with associativity gives the equation:

$$g(x, u) \mathbin{@} z = g(x, u \mathbin{@} z) \tag{$**$}$$

Critical pair computations with Rules 3, and Rule $(*)$ return pairs: $(nil \mathbin{@} u, g(nil, u))$, and $(g(xs, [x]) \mathbin{@} u, g(x :: xs, u))$. The left-hand side of the first pair reduces into u by rewriting with the left unit law, $nil \mathbin{@} x \rightarrow x$. The left-hand side of the second pair reduces by rewriting with equation $(**)$ conveniently oriented into the rule $g(x, u) \mathbin{@} z \rightarrow g(x, u \mathbin{@} z)$. The result is $reverse(xs) \mathbin{@} ([x] \mathbin{@} u)$, which further reduces with the synthesis rule itself into $g(xs, [x] \mathbin{@} u)$. The system has discovered the definition of g:

$$g(nil, u) \rightarrow u \tag{5}$$
$$g(x :: xs, u) \rightarrow g(xs, [x] \mathbin{@} u) \tag{6}$$

which is tail recursive. Use of another law of @: $[x] \mathbin{@} y \;\rightarrow\; x :: y$, reduces the left-hand side of rule 6 into $g(x :: xs) \;\rightarrow\; g(xs, x :: u)$. Now the definition of *reverse* does not refer to @:

$$reverse(x) \;\rightarrow\; g(x, nil) \qquad g(nil, u) \;\rightarrow\; u \qquad g(x :: xs, u) \;\rightarrow\; g(xs, x :: u)$$

This derivation is replicated each time a recursive call occurs under an associative operator with left and right unit. A more sophisticated instance of this strategy eliminates one recursive call in the following example [21], originally proposed by P. Chatelin [7].

$$\begin{aligned} height(tip(x)) &\;\rightarrow\; 0 \\ height(tree(lt, rt)) &\;\rightarrow\; 1 \;+\; max(height(lt), height(rt)) \end{aligned}$$

Here the recursive calls occurs under two associative operators with left and right unit, namely $(+)$ and *max*. Moreover the operator $+$ distributes over *max*, allowing use of the associativity of *max*.

6 Generating implementations

Following several stages of transformation, our system produces a first-order SML program that is functionally equivalent to the computational semantics of a sentence in the DSDL that a user has written. This program can be compiled by an SML compiler to produce an executable software module. To execute this module, the run-time support for SML needs to be present, however. It is often the case that the requirements of a software architecture, of a target platform for the software, or of standards adopted by a software organization dictate another implementation. To provide for alternate implementations, a back-end tool called the *Program Instantiator* generates target code to meet requirements imposed on a desired implementation.

The Program Instantiator (abbreviated PI) is based upon earlier research by Dennis Volpano [25, 26]. It is driven by several parameters of an implementation, which include:

- the target programming language in which an implementation is to be coded;
- templates in the target language that realize implementations of the concrete algebras used in a program;
- target language templates that provide a standard implementation of free term algebras and of the case discrimination on data constructors;
- templates for function calls and module headers in the target language.

The PI also interprets an environment specification that provides the types and structure of data and control interfaces with a host software architecture. The output of the PI is a module (or modules) in the syntax of the specified target language that provides an implementation of the first-order SML program given it as input. The PI is currently the least mature of the tools in the translation pipeline and several issues remain to be resolved. These include:

* duplicate function declarations. There is currently no test for function definitions that are identical, up to renaming, and hence could be identified.
* heap storage management. The PI does not currently generate a general-purpose garbage collector. It performs storage allocation in blocks that can be collected in total if lifetimes of data are known to be limited.

* special scoping restrictions. Some possible target languages ('C', for instance) impose restrictions on the declarations of nested scopes. The PI does not currently provide for such restrictions.

7 Implementing the pipeline

The translation and transformation tools described in the preceding sections have all been implemented in Standard ML (SML) [19] except for Astre, the term-rewriting transformation tool, which is implemented in CAML. Furthermore, a restricted sublanguage of SML is used for the intermediate representation of programs as they are passed through the pipeline. An abstract syntax representation of SML is used internally by each tool. This representation is unique to the SDRR tools and has little in common with the internal representation used by the SML/NJ compiler, for instance.

Use of SML language technology has been an important factor in the success of the fifteen-month project in which most of the tool development occurred. It has allowed considerable code reuse among tools, and has simplified integration and testing procedures.

7.1 Common parts

The suite of tools shares several common parts. These tools communicate program images in a restricted dialect of SML. Internally, most of the tools use a common representation for the abstract syntax of SML terms. They share a common parser and prettyprinter to destructure and reconstitute the textual image of a program.

Type reconstruction is required for order reduction, for postprocessing by Astre and by the program instantiator. However, each of these tools needs some additional information or needs typing information presented in a different format. We have used a common type inference tool that is customized with an output module to meet the needs of its various clients.

Reuse of these common parts has contributed substantially to the ease with which the family of tools could be maintained and kept consistent.

8 An application generator

The SDRR method has been applied to design a software component generator for message translation and validation (MTV). This is an application that arises in military command and control systems, with automatic teller machines in banking and with point-of-sale terminals for retail stores. A central controller receives byte-string encoded messages from remote sensors or terminals. It must validate each message and translate it into an internal format for further analysis and response. A controller may serve several sensors, each of which generates messages in a different format. An MTV module for a particular message format analyzes a string of bytes given as input to check whether it has the expected structure, reports errors if the input is not a valid message, and translates the input into a data structure representing the contents of the message if the input is valid. A system must include an MTV module for each message type that it expects to receive. Thus a generator that is capable of interpreting an message format description and producing an MTV module for the specified message type is useful.

A message specification is presented to an engineer as an *interface control document* (ICD). The ICD is a semi-formal description of the string-encoded format of the message. It consists of general information, such as the expected length of a message, followed by a field-by-field description of its contents. Field descriptions may themselves have internal structure. For example, a date field will contain a day, month and year. Some fields may represent various types of data. For example, a field may represent an altitude if it contains only digits or a location if it contains alphabetic characters. The ICD also contains constraints on valid messages; these are expressed informally in natural language. We have designed a Message Specification Language (MSL), which is a domain-specific design language for the MTV application.

For the MTV domain the essential abstractions are the internal and external representations of messages. They are related by translation functions that map between them. A logical representation in which both intra- and inter-field constraints are imposed is introduced as an intermediate representation. From the logical representation, a controller can derive the necessary internal representation. There is also a "user" representation, which is an Ascii string in a format readable by humans, that is used for logging messages received by a controller or for manual entry of a message.

A software module for MTV consists of six components:

- two functions that check the formats of external or user messages,
- two functions that translate between external and internal formats, and
- two functions that translate between user and internal formats.

The MSL language describes the logical structure of the internal representation of data, the message translation action that parses a message, scaling of numeric values, and any constraints imposed on the values in fields of the message. From these descriptions, the MSL translator and the SDRR transformation pipeline generate the six components of the solution as an Ada package.

8.1 The Message Specification Language

To use the MTV generator, an engineer specifies the logical structure of a message as a logical type in MSL. In the example that follows, square brackets enclose the components of a labeled sum and curly brackets enclose the components of a labeled product. Instances of labeled products are record structures; labeled sums are types for variant records.

```
(* Type declarations *)
type Confidence_type = [High, Medium, Low, No];

type Alt_or_TC_type = [Altitude: integer(1..99),
                       Track_confidence: Confidence_type,
                       No_value_or_Alt_less_than_1000];

type Time_type = {Hour: integer(0..23),
                  Minute: integer(0..59)};

message_type MType = {Course: integer(0..360),
                      Speed: integer(0..5110),
                      Alt_or_TC: Alt_or_TC_type,
                      Time: Time_type};
```

The basic types used in messages of this type are integers and integer subranges. These are arranged into labeled sums (`Confidence_type` and `Alt_or_TC_type`) and labeled products (`Time_type` and `MType`).

The engineer also specifies the translation map in one direction: from external to logical. This specifies an external message reader (EXR). For example:

```
(* Action declarations *)
EXRaction to_Confidence = [High: Asc 2 | "HH",
                           Medium: Asc 2 | "MM",
                           Low: Asc 2 | "LL",
                           No: Asc 2 | "NN"];

EXRaction to_Alt_or_TC = [Altitude: Asc2Int 2,
                          Track_confidence: to_Confidence,
                          No_value_or_Alt_less_than_1000: Skip 0
                         ] @ Delim "/"; (*  field separator "/" *)

EXRaction to_Time = {Hour: Asc2Int 2,
                     Minute: Asc2Int 2
                    } @ Delim "\r";     (* CR as field separator *)

EXRmessage_action to_MType = {Course: Asc2Int 3 @ Delim "/",
                              Speed: Asc2Int 4 @ Delim "/",
                              Alt_or_TC: to_Alt_or_TC,
                              Time: to_Time};
```

Message reader declarations are a fundamental syntactic construct in MSL, and are given semantics in its formal definition. The semantics is compositional and makes use of the structure implicit in the types declared for the corresponding fields, Primitive translation functions such as `Asc2Int` provide a basis for the translation actions. For example, `Asc2Int 2` reads two Ascii characters (which must be numerals) and produces an integer value. Specified reader actions for individual fields can be aggregated into a record reader by enclosing them in curly braces.

To accommodate variant record readers, a failure/backtrack mechanism is provided in the semantics of MSL. If the data string presented the reader action `EXRaction to_Alt_or_T` is a sequence of digits, it is interpreted to denote an altitude, but if it is alphabetic it is a track confidence. If it is neither of these, it must be empty. In all cases the field is delimited by a slash character. The semantics of an external message reader implements a parser for a simple language without recursion in its grammar.

From the specification of an external message reader, the MSL translator infers the inverse mapping from logical to external representation and also the logical to user mappings. For either the external to logical or the user to logical translation, the semantics must prescribe checking of constraints on values of fields in the message. Constraints are of two kinds:

- Subrange specifications on an individual field. These are specified in a field type and are translated as range checks;
- Inter-field dependencies. These can involve conjunctions or disjunctions of boolean-valued expressions that refer to values in different fields.

An MSL specification is declarative, rather than algorithmic. Maintenance of an artifact expressed at this level is expected to be significantly easier than maintenance of a code level representation. A graphical user interface is used to help application engineers formulate or modify a message design in the MSL language.

9 Conclusions

We have successfully demonstrated an automated transformation system that compiles practical software modules from the semantic specification of a domain-specific application design language. The integrated suite of transormation and translation tools represents a new level of design automation for software. Although there is much more that can be done to further improve the performance of generated code, the prototype system demonstrates the feasibility of this approach.

The implementation of type-parametric theorems as transformation tactics for HOT has not been done before. It remains to be seen whether algebra-specific transformations can be incorporated in the same tool by refering to a database of algebraic laws. In the current system, algebra-specific transformations are performed by term-rewriting, which is an entirely different paradigm.

Acknowledgements

We wish to acknowledge the generous help of Andrew Tolmach, who shared with us his extensive knowledge and valuable insight of Standard ML language technology, and of Satnam Singh, who furnished expert advice on Ada code generation issues.

References

[1] Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1994.

[2] F. Bellegarde. A transformation system combining partial evaluation with term rewriting. In *Higher Order Algebra, Logic and Term Rewriting (HOA '93)*, volume 816 of *Lecture Notes in Computer Science*, pages 40–58. Springer-Verlag, September 1993.

[3] F. Bellegarde. Termination issues in automated syntheses. Submitted to RTA95, September 1994.

[4] Françoise Bellegarde. Program transformation and rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 226–239, Berlin, 1991. Springer-Verlag.

[5] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO Series F*. Springer-Verlag, 1986.

[6] W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[7] P. Chatelin. Program manipulation: to duplicate is not to complicate. Technical report, Université de Grenoble, 1976. Technical Report CNRS, Laboratoire d'informatique.

[8] W. Chin and J. Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

[9] Wei-Ngan Chin. Safe fusion of functional expressions. In *Proc. of 1992 ACM Conf. on Lisp and Functional Programming*, pages 11–20, June 1992.

[10] J. Darlington and R. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[11] N. Dershowitz. Synthesis by completion. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 208–214, Los Angeles, 1985.

[12] Nachum Dershowitz. Completion and its applications. In *Resolution of Equations in Algebraic Structures*. Academic Press, New York, 1988.

[13] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Twente, The Netherlands, February 1992.

[14] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In J-P. Jouannaud, editor, *Functiona Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer Verlag, 1985.

[15] Richard B. Kieburtz and Jeffrey Lewis. Algebraic Design Language—Preliminary definition. Technical report, Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, January 1994.

[16] Grant Malcolm. Homomorphisms and promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, June 1989.

[17] Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In *Proc. of the CWI Symbposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[20] Peter D. Mosses. Sis—semantics implementation system: reference manual and user guide. Technical Report DAIMI MD-30, Computer Science Department, University of Aarhus, 1979.

[21] A. Petorossi and M. Priotti. Rules and strategies for program transformation. In *Formal Program Development, IFIP TC2/WG2.1 State of the Art Report*, volume 755 of *Lecture Notes in Computer Science*, pages 263–304. Springer Verlag, 1994.

[22] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[23] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.

[24] Timothy Sheard. Optimizing algebraic programs. Technical Report OGI-CSE-94-004, Oregon Graduate Institute of Science & Technology, January 1994.

[25] Dennis Volpano and Richard B. Kieburtz. Software templates. In *Proceedings Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society, August 1985.

[26] Dennis Volpano and Richard B. Kieburtz. The templates approach to software reuse. In Ted J. Biggersstaff and Alan J. Perlis, editors, *Software Reusability*, pages 247–255. ACM Press, 1989.

[27] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *2nd European Symposium on Programming ESOP'88*, volume 300 of *LNCS*. Springer-Verlag, 1988.