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

1 A technology for automatic program generation

Program generators can substantially reduce the effort needed to produce versions of a common software design that are tailored to particular applications, 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.

For a DSDL to be used to express input to a program generator, it must have 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 computability requires semantic

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

functions to be expressed algorithmically. Requiring totality allows the use of 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 [19] 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 can 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 properties of the operations of a particular computer architecture. Such transformations typically occur in the code generator of an optimizing compiler.

A compositional programming style introduces many intermediate data structures. When semantic functions are applied directly, their compositions may entail multiple traversals of data structures that represent the abstract syntax of the object language. 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 [25, 8], and
- the tupling, or parallel fusion strategy [6, 9], in which a pair of functions that operate on the same data are transformed into a single function that returns a result pair. 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 when formulating semantic functions and if this information is preserved through the translation process, it can be exploited to drive 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 [9], 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 [11, 2]. 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 improvement 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. Specialization may increase the size of a program but has no negative impact on its execution time, and often improves it. Generation of an appropriate data structure to represent closures [20, 1] leads to a more general but less straightforward approach for order-reducing transformations.

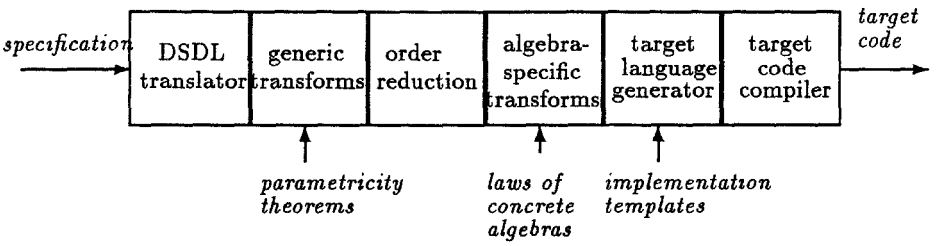


Figure 1—Transformation and translation pipeline

3 Computable denotational semantics

Denotational semantics for programming languages interpret syntax by means of 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. To

ensure that the semantics of a specification language is computable, its intuitive meaning is formalized in terms of an executable meta-language.

We have designed the ADL language [14] 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, 16, 17, 12] to derive programs from logical specifications by formal reasoning. In ADL, structure algebras are first-class entities that can be declared, bound to identifiers and form the basis for ADL control operators. The declarative elements of ADL include *signatures* of algebraic varieties, *algebra specifications* and constant (value) declarations.

Signature declarations do not use explicit recursion, for a signature defines not just a single algebra, but an entire class (or variety) of algebras that share a common structure. For example, the signature declaration 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.

An algebra specification binds a type for the carrier and a compatibly typed constant for each operator symbol. An example of a *list*-algebra specification is:

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

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 whose carrier is 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 [18]. For each variety declared by a signature in an ADL program, its free term algebra functor is implicitly declared.

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 only its operators and their typings are known. 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. For example, the type *int* is the carrier of a concrete algebra of integer arithmetic, which is externally specified.

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 [2]. 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 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

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

denotes a function that sums the elements of a list of integers. This function is an example of a *list*-algebra homomorphism; the condition that it satisfies is

$$sum\ Nil = 0$$
$$sum\ (cons(x, y)) = x + sum\ y$$

Had *sum* been defined in a language such as SML using explicit recursion, then the homomorphism equations would constitute its declaration. However, recursion is not explicit in ADL, it is instead calculated from the signature declaration given for the variety *list*. The combinator *red* has also been called a catamorphism combinator [17].

[2] 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.

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 [15]. 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 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

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

where the expression enclosed in curly brackets denotes the operation of 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

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

where $\mathbf{in}^{list}$ is the composite operator of the free *list* algebra.

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

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

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

which can be read from the commuting diagram:

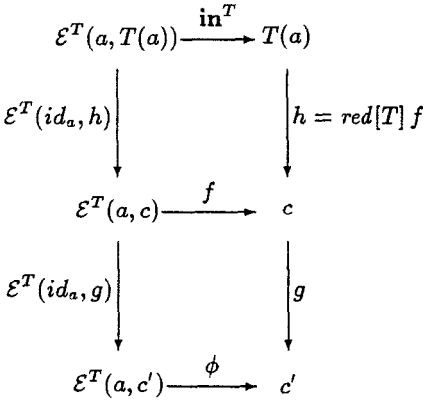
$$\begin{array}{ccc} \mathcal{E}^{list}(a, list(a)) & \xrightarrow{\mathbf{in}^{list}} & list(a) \\ \Big\downarrow \mathcal{E}^{list}(id_a, h) & & \Big\downarrow h = red[list]\{c;\ f_{nil},\ f_{cons}\} \\ \mathcal{E}^{list}(a, c) & \xrightarrow[\{f_{nil},\ f_{cons}\}]{} & c \end{array}$$

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

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 the algebra $\{c'\ \phi\}$.



□

The higher-order transformation tool, HOT, uses a clever heuristic tactic to calculate an operator ϕ that satisfies the promotion theorem [21, 22]. 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 occurrence 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 [13], which removes nested function declarations and explicit abstractions, replacing them by new, closed function declarations. 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 add dummy arguments to an applicative expression that is unsaturated.
  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 [7], 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 [7]). 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 an algorithm due to Reynolds [20].

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*:

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

with respect to the constant *sqr*, obtaining

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

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. Reynolds' algorithm constructs a closure representation for higher-order functions that are not variable-or-constant only.

5 Algebra Specific Transformations

Many transformations are justified in part by the laws of specific algebras. As a logical extension to an ADL module, properties of an imported 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 needs attention in the future development of our system.

Common equational laws such as associativity, commutativity, distributivity, unit laws and inverse laws can justify tactics such as recursion elimination, which can sometimes reduce the asymptotic complexity of an algorithm.

Astre is a transformation tool based on rewriting techniques [3]. 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 available 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 allows both folding and unfolding of definitions, depending upon the orientation of the equations as rules. Critical pair computation performs both instantiation and unfolding and hence can implement transformations by the unfold/fold technique. This has been called *synthesis by completion* [10, 11].

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 in which functions are presented by a set of mutually recursive equations with pattern-matching arguments.

A fully automated transformation system needs additional techniques, including:

- a mechanism that introduces new function definitions to form *synthesis rules*. 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 orients critical pairs into rules as required by the transformation strategy. It guarantees that termination of the rewrite system is

preserved during the synthesis. Astre carefully controls the production of critical pairs to ensure that the completion process does not diverge [4].

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 (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 (4), yielding $reverse(x :: xs) \rightarrow g(xs, [x])$. 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 (3) and $(*)$ 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 (6) into $g(x :: xs) \rightarrow g(xs, x :: u)$. The new definition of *reverse* no longer refers 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.

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. Often, the requirements imposed by a software architecture, a target platform for

the software, or standards adopted by a software organization dictate a specific form of 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 [23, 24]. 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 implements 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 entirely when the data they contain are no longer accessible.
* 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) [18] 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 transformation pipeline 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 project during the fifteen months in which most of the tool development occurred. It has allowed substantial code reuse among tools, and has simplified integration and testing procedures.

8 An application generator

The design method we have described here has been applied to design a software component generator for message translation and validation (MTV). This application 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 messages encoded as byte-strings 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 is required for each message format. It 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.

Under current practice, an engineer receives a message specification in the form of an *interface control document* (ICD). An ICD is a semi-formal description of the string-encoded format of the message. It gives the maximum 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. A field may represent various types of data. For example, a field may represent an altitude if it consists only of digits or a location if it contains alphabetic characters. An ICD can also specify constraints on valid messages; these are expressed informally in natural language. We have designed a Message Specification Language (MSL), which is a formal, 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. It 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 a message, the translation action that parses a message, scaling of numeric values, and any constraints imposed on the values of fields. From these descriptions, the MSL translator and the transformation pipeline generate the required six functions 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. Labeled sums are types for variant records. Labeled products are types for records, but they are not illustrated here.

```
(* 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];
```

The engineer also specifies the translation map in one direction: from external to logical. This specifies an external message reader (EXR). For the field types shown above, the external reader declarations are:

```
(* 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 "/" *)
```

Message reader declarations are a fundamental syntactic construct in MSL, and are given semantics in its formal definition. The semantics makes use of the structure implicit in the types declared for the corresponding fields. Primitive translation functions such as `Asc2Int` provide basic translation actions. For example, `Asc2Int 2` reads two Ascii characters (which must be numerals) and produces an integer value.

From the specification of an external message reader, the MSL translator not only compiles a message parser that produces a logical representation, but also infers the inverse mapping from logical to external representation and 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.

A generator for MTV modules has been implemented with the technology described in this paper and evaluated in an experiment whose results will be reported elsewhere.

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 transformation 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 referring 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. We are grateful to Laura McKinney and Alexei Kotov, who provided management and measurement of our project, essential to its success.

References

1. J. M. Bell and J. Hook. Defunctionalization of typed programs. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, Feb. 1994.
2. F. Bellegarde. Program transformation and rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, vol. 488 of *LNCS*, pages 226–239, Berlin, 1991. Springer-Verlag.
3. F. Bellegarde. A transformation system combining partial evaluation with term rewriting. In *Higher Order Algebra, Logic and Term Rewriting (HOA '93)*, vol. 816 of *Lecture Notes in Computer Science*, pages 40–58. Springer-Verlag, Sept. 1993.
4. F. Bellegarde. Termination issues in automated syntheses. Submitted to RTA95, Sept. 1994.
5. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, vol. 36 of *NATO Series F*. Springer-Verlag, 1986.
6. W. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
7. W. Chin and J. Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.
8. W.-N. Chin. Safe fusion of functional expressions. In *Proc. of 1992 ACM Conf. on Lisp and Functional Programming*, pages 11–20, June 1992.
9. J. Darlington and R. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.
10. N. Dershowitz. Synthesis by completion. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 208–214, Los Angeles, 1985.

11. N. Dershowitz. Completion and its applications. In *Resolution of Equations in Algebraic Structures*. Academic Press, New York, 1988.
12. M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Twente, The Netherlands, Feb. 1992.
13. T. Johnsson. Lambda lifting: transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functiona Programming Languages and Computer Architecture*, vol. 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer Verlag, 1985.
14. R. B. Kieburtz and J. Lewis. Algebraic Design Language—Preliminary definition. Technical report, Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, Jan. 1994.
15. G. Malcolm. Homomorphisms and promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, vol. 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, June 1989.
16. L. 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.
17. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, vol. 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, Aug. 1991.
18. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.
19. P. D. Mosses. Sis—semantics implementation system: reference manual and user guide. Technical Report DAIMI MD-30, Computer Science Department, University of Aarhus, 1979.
20. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.
21. T. Sheard. Optimizing algebraic programs. Technical Report OGI-CSE-94-004, Oregon Graduate Institute of Science & Technology, Jan. 1994.
22. T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.
23. D. Volpano and R. B. Kieburtz. Software templates. In *Proceedings Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society, Aug. 1985.
24. D. Volpano and R. B. Kieburtz. The templates approach to software reuse. In T. J. Biggersstaff and A. J. Perlis, editors, *Software Reusability*, pages 247–255. ACM Press, 1989.
25. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *2nd European Symposium on Programming ESOP'88*, vol. 300 of *LNCS*. pages 344-358, Springer-Verlag, 1988.