Automatic Generation of Execution Tools in a GANDALF Environment*

Vincenzo Ambriola ** **and Carlo Montangero**
Dipartimento di Informatica, Universita' di Pisa, Pisa, Italy

The formal definition of a programming language in denotational style is taken as the basis for the automatic generation of its interpreter. The facilities available in GANDALF are exploited to implement and integrate such a generation technique in a GANDALF environment.

1. INTRODUCTION

There is a widespread interest in exploiting the formal definition of a programming language in the design and the implementation of a variety of programming tools. This approach, originally developed in the implementation and validation of compilers, has been recently applied to the design of programming environments. Initial results include the successful generation of structure editors, automatically driven by the syntactic description of a language, and the generation of static semantics tools based on the use of attribute grammars. It is likely that further results will soon be achieved applying the same guidelines that made the production of a compiler a well understood task.

The work reported in this paper shows how the interpreter for a programming language can be automatically derived and integrated into a GANDALF environment, starting from the denotational semantics of the language. The derivation of the interpreter is based on the transformational approach [1, 2, 3]. Semantics-preserving transformations are applied to the denotational definition of the language in order to obtain a simpler but equivalent definition that can be translated into a C program [4]. The compilation of this program finally produces the interpreter.

The generation of the GANDALF environment, in which the interpreter is available as an integrated tool, is achieved in three phases:

- A *semantic* phase that generates the interpreter and the abstract syntax of the language in ALOEGEN notation;
- A *syntactic* phase that generates a purely syntactic editor, according to the abstract syntax generated in the semantic phase;
- A *merge* phase that integrates the purely syntactic editor and the interpreter.

In the semantic phase, denotational semantics is given using a structure editor for the formal notation. This editor is able to perform the transformations needed to complete the phase. The syntactic phase is a standard editor generation performed using ALOEGEN. The last phase consists of the extension of the standard set of commands, provided by ALOEGEN for the editor generated in the syntactic phase, with a new command corresponding to the interpreter invocation.

In our proposed solution we have achieved a twofold goal. We generate a GANDALF environment in which the interpreter is an integrated tool, and we use another GANDALF environment for the definition and the transformation of denotational semantics. This demonstrates that a GANDALF environment is powerful enough to support the entire generation process and also flexible enough to allow the integration of an execution tool derived from a formal definition.

There are also other advantages to the use of a GANDALF environment. The meta-environment for the definition of denotational semantics has been

* This work has been supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, N.J. and by Progetto Finalizzato Informatica del CNR Obiettivo CNET.

** Vincenzo Ambriola is on leave from Dipartimento di Informatica, Universita' di Pisa, Italy and is a visiting scientist at CMU.

Address Correspondence to Vincenzo Ambriola, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA. 15213

The Journal of Systems and Software 5, 155–171 (1985)

0164-1212/85/$3.30

generated as a standard GANDALF programming environment. The transformations have been implemented by using the action routine model and the unparsing facilities provided by ALOEGEN. The interpreter, generated according to the action routine model, has direct access to the internal program representation and exploits auxiliary semantic structures automatically maintained by the editor kernel. The final environment allows direct interpretation of programs, thus avoiding the need of a compilation cycle. A straightforward extension of the interpreter generation will make possible execution of incomplete programs and generation of sophisticated debuggers.

Section 2 describes the actual trend in tool generation for programming environments. Section 3 gives a brief overview of the generation process. Section 4 introduces the metalanguage that we use for defining denotational semantics. Sections 5 and 6 detail the transformations that produce the interpreter. An example is carried out throughout the paper.

2. TOOL GENERATION IN PROGRAMMING ENVIRONMENTS

The task of developing a programming environment is more easily achieved if it is based on a declarative description of the objects involved in the programming activity. Indeed, the need for a declarative style is usually combined with the requirement of a higher level specification of the desired environment. Using a declarative approach leads to the construction of powerful meta-tools, also called generators, that take as input the formal description of a language and automatically produce a complete, often optimized, programming tool. In its extreme application, meta-tools can be used to produce other generators recalling the bootstrapping process widely used in computer systems.

The validity of this strategy was experimented with early in the GANDALF project. The actual syntax editor generator, ALOEGEN, is itself a structure editor described and generated in the same way as other ALOE editors. In this case, the declarative description of the abstract syntax and of the user interface has been the keystone for the effective production of a variety of GANDALF environments.

After the initial results achieved in the generation of syntactic tools, the logical next step was to consider the description and the automatic generation of semantic tools. In particular, this second generation of tools was divided into two broad classes:

1. Tools for static semantics;
2. Tools for dynamic semantics.

The former is also called the class of context-sensitive tools and comprises symbol processors, type-checkers, and scopers. In a wider sense, this class can be extended to cover most of the user interface aspects of a programming environment. The latter class contains execution tools like interpreters, debuggers, code generators, and compilers. In this paper we will concentrate our attention on this latter class.

The initial efforts have been focused on the solution of problems related to the class of tools for static semantics. Attribute grammars [5, 6, 7] have been revised and adapted in order to fit the efficiency requirement that is critical in interactive systems. Parameterized modules are another generative approach exploited in the Pecan system [8] and derived from the author's previous experience in compiler compiler technology [9].

Different generation techniques, including action routines [10, 11, 12], are more operational in style. The semantic processing is described as a set of routines in a conventional programming language. Each routine is automatically activated by the editor when the edited program changes. The drawback of this approach is clearly the lack of a formal and declarative description of the semantic processing. On the other hand, the resulting tools are well-suited for coping with the interactive and incremental nature of static semantics in programming environments.

Starting from a different notion of programming environments, where incrementality is not the principal requirement, another approach to the generation of programming tools has been proposed in the CNET project [13]. Here a complete environment is generated as a collection of tools derived from the denotational definition of a programming language. Static semantics tools, in this case, are derived using non-standard semantics [14, 15, 16]. In another system, the Programming System Generator [17], denotational definitions and attribute grammars are coupled together to describe the semantics of the generated environments.

The state of the art for specification and generation of tools for dynamic semantics is completely different. In all the above mentioned systems, except those making use of denotational definitions, the generation of execution tools is generally achieved with a low and often lacking degree of formality. Typically, dynamic tools are pre-existing compilers, code generators, and interpreters defined *ad hoc* for a specific language. Specific, rather than generic, solutions for dynamic tools are usually rationalized by looking to the strong interaction of these tools with

the environment or to the issue of their efficiency.

Coming back to the semantic model proposed in the GANDALF system, it is worth noting that the action routine model has the advantage of allowing efficient implementations of static and semantics tools. However, in the general economy of the GANDALF project, the action routine model is the final target of a more declarative solution, where the semantics of a programming language directly drive the generation of a wide class of tools. Work in this direction is already going on with respect to a declarative specification of static semantics [18]. The work reported in this paper is another attempt to introduce a declarative and formal method in the automatic generation of GANDALF environments.

3. A BRIEF OUTLINE OF THE GENERATION PROCESS

Before entering into the numerous details of its description, in this section we give a brief outline of the generation process. In our first attempts to understand this process, we learned that the best vehicle for such an explanation was a figure containing the functional components of the system and their logical and temporal connections.

Figure 3-1 is divided in two parts. In the upper part we have described the generation of the meta-environment that we have built once for all and that is used by the implementor of a programming language. In the lower part we have described the generation of a specific environment for a programming language, obtained using the meta-environment. The objects in round boxes represent instances of ALOE; the objects in square boxes represent other tools.

The meta-environment is an ALOE for the metalanguage MELA, extended with a set of action routines and extended commands. It has been generated using the GANDALF editor generator ALOEGEN. MELA is the metalanguage that we have defined to give the denotational semantics of programming languages. It is presented and discussed in Section 4. The ALOE for MELA performs a complete static analysis of the semantic definitions and the transformations required by our generation process.

The implementor of the environment for a programming language L uses the MELA editor for defining the semantics of L. These definitions are then tested and transformed by the editor into equivalent ones. The transformation phase is called *defunctionalization* because all the higher order functions introduced in the denotational semantics of L are removed and replaced. The details of this phase are given in Section 5.

At the end of this phase the editor produces the abstract syntax for the language L and the source code of the interpreter. The source code is compiled and used to extend the ALOE generated by ALOEGEN, according to the abstract syntax of L. The abstract syntax and the source code of the interpreter are produced using the unparsing facilities provided by ALOEGEN. The details are given in Section 6.

To implement this simple technique we have to satisfy a single requirement: the mapping from MELA to C must be context-free. This motivates the need for removing the higher order functions that, otherwise, cannot be directly mapped into C constructs. Once that we have removed this major difficulty, it has been easy to define a context-free mapping for the metalanguage.

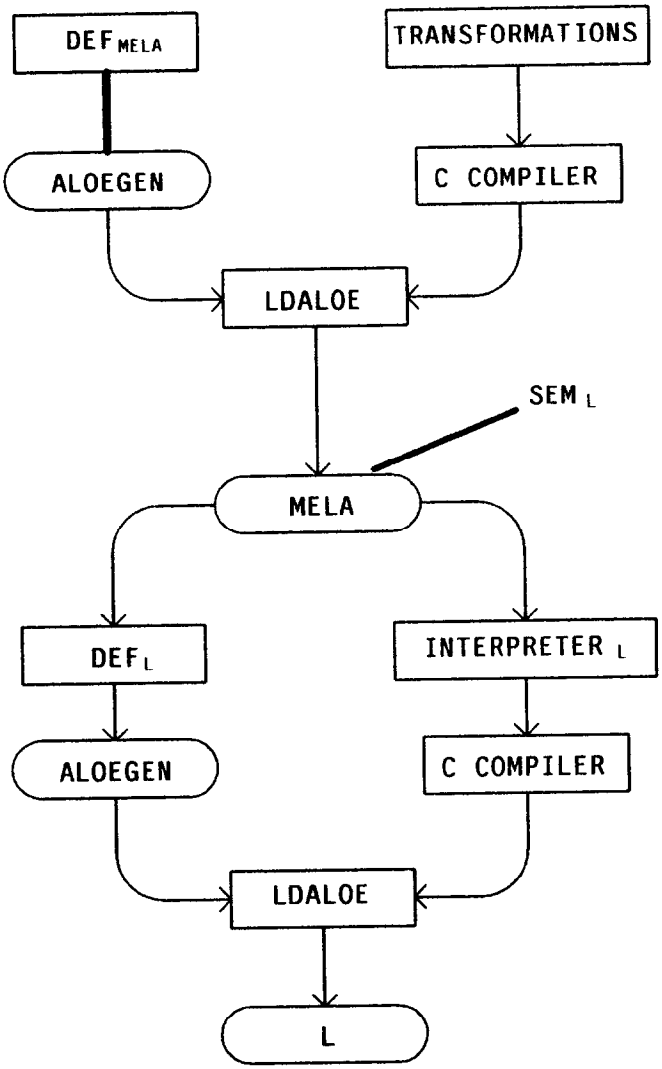

Figure 3-1: Generation process

4. THE METALANGUAGE

In this section we introduce the metalanguage (MELA) used to define the syntax and the semantics of a programming language. In the next two sections we show how definitions given in MELA can be automatically transformed into C programs. In other words, we will give the operational semantics of MELA.

MELA is a strongly typed functional language, which allows the definition of higher order functions and domains. It has been designed having in mind its primary use as a metalanguage for specifying denotational semantics of programming languages.

A denotational definition of a programming language, *i.e.* a program in MELA, consists of three parts: the first and the second part define the syntactic and semantic domains involved in the definition of a programming language. The third part contains the semantic functions that constitute the core of the semantic definition. The abstract syntax (the syntactic domains in MELA) describes the structure of valid programs for the language; the semantic domains describe the data structures for the semantic concepts of the language. Each semantic function is defined by cases: one case for each component of the abstract syntax. Auxiliary semantic functions are usually required for dealing with semantic domains.

The complete syntax of MELA is given in Appendix I. A discussion of the main features of MELA, informally introducing its semantics, is given in the rest of this section. In order to make more concrete the description of the metalanguage we use a sample language T taken from [19]. The syntax of this language is shown in Figure 4-1. Its semantics will be given at the end of this section.

```
⟨prog⟩ ::= prog ⟨com⟩
⟨com⟩  ::= skip |
           ⟨com⟩; ⟨com⟩ |
           if ⟨expr⟩
             then ⟨com⟩ fi |
           while ⟨expr⟩
             do ⟨com⟩ od |
           ⟨ide⟩:= ⟨expr⟩ |
           write ⟨expr⟩
⟨expr⟩ ::= ⟨bool⟩ |
           ⟨int⟩ |
           ⟨ide⟩ |
           neg (⟨expr⟩)
⟨bool⟩ ::= true |
           false
⟨int⟩  ::= ⟨number⟩
```

Figure 4-1: BNF of the sample language T

In the remainder of the paper, we assume that the reader is familiar with denotational semantics of programming languages, as presented in [20, 21]. This knowledge is not essential for the understanding of the generation technique, which is based on a set of program transformations, but it is useful for a *semantic* understanding of the mathematical notions involved in the presented matter.

4.1. Domains

MELA has three primitive flat domains: the domains of boolean values (Bool), the domain of integer values (Int), and the domain of identifiers (Ide). Non-primitive domains are introduced by the following composition rules[1]:

Enumeration: $D = \{C_1, \ldots, C_n\}$. The elements of D are the constants $C_1, \ldots, C_n$ ($n > 0$). Constants belong to the domain Ide of identifiers;

Disjoint Union: $D = D_1 + \ldots + D_n$. The elements of D are those of $D_1, \ldots, D_n$ ($n > 1$). The addends of the (coalesced) union must have different names and cannot be disjoint unions themselves;

Cartesian Product: $D = D_1 : P_1 \times \ldots \times D_n : P_n$. The elements of D are tuples of elements from $D_1, \ldots, D_n$ ($n > 0$). $P_1, \ldots, P_n$ are projector identifiers used to select the components of a D tuple;

Function Space: $D = D_1 \times \ldots \times D_n \rightarrow D_{n+1}$. The elements of D are the continuous functions from $D_1 \times \ldots \times D_n$ into D_{n+1} ($n \geq 0$).

In MELA there is no explicit way to define a new domain equivalent to another domain: the definition of a cartesian product with a single factor has to be used to mimic such a definition.

As we have already mentioned, domains can be defined in the syntactic and semantic part of a MELA program. In each part domain definitions are mutually recursive. However, domains introduced in the syntactic part can be used in the definition of semantic domains, while the reverse is not true.

According to the ALOEGEN notation used for the abstract syntax definition, enumerations are used in the syntactic part to define terminal operators, cartesian products to introduce non-terminal operators, and unions to introduce classes. Functional domains cannot be defined in this part as they would have no meaning.

---

[1] Throughout the paper, D will be used as a metavariable ranging over *Dname*, C as a variable ranging over *Constname*, and P ranging over *Projname*.

As an example, consider the syntactic domains ***PROG***, ***COM***, and ***EXPR*** of the language T. The corresponding syntactic MELA domains are given in Figure 4-2.

```
PROG   = COM: Com

COM    = SKIP + SEQ + COND +
         LOOP + ASSIGN + WRITE

SKIP   = {skip}

SEQ    = COM: SeqCom1 x
         COM: SeqCom2

COND   = EXPR: CondExpr x
         COM: CondCom

LOOP   = EXPR: LoopExpr x
         COM: LoopCom

ASSIGN = IDE: AssignIde x
         EXPR: AssignExpr

WRITE  = EXPR: WriteExpr

EXPR   = BOOL + INT +
         IDE + NEG

NEG    = EXPR: NegExpr
```

Figure 4-2: Syntactic domains of T in MELA

The semantic domain part introduces the domains that are needed to give the denotational semantics of a language. In general they are functional domains, most likely to be used as auxiliary domains. The semantic domains used to give the denotational semantics of the language T are given in Figure 4-3.

$V = INT + BOOL$

$U = V + \{unbound\}$

$O = \{nil\} + (V \times O)$

$M = IDE \rightarrow U$

$S = M \times O$

$C = S \rightarrow S$

Figure 4-3: Semantic domains of T

V and ***U*** are two simple domains that define the denotable values of T. ***O*** and ***M*** model output stream and memory, respectively. ***S*** is a cartesian product of ***M*** and ***O*** and corresponds to the state of a program during its execution. ***C*** is a more complex domain that will be used to give the semantics of loop statements. The corresponding semantic domains are given in MELA in Figure 4-4.

```
VAL    = INT + BOOL

UNB    = {unbound}

UVAL   = INT + BOOL + UNB

NIL    = {nil}

OUT    = NIL + STREAM

STREAM = VAL: Last x
         OUT: Prev

MEM    = IDE -> UVAL

STATE  = MEM: Mem x
         OUT: Out

CMD    = STATE -> STATE
```

Figure 4-4: Semantic domains of T in MELA

As a final comment, it is worth noting that MELA syntactic and semantic primitive domains are equivalent. As a consequence, integer numerals and integer numbers collapse into a single domain Int. It will be shown in the sequel that the part of the interpreter dealing with these domains exploits this equivalence.

4.2. Semantic Functions

Semantic functions are given as mutually recursive function definitions of the form:

$F(D_1: V_1; ...; D_n: V_n): D <= E$

where F is a metavariable ranging over *Funname*, V ranges over *Varname*, and E ranges over *Expr*.

The main syntactic categories in MELA are (structure) expressions *Expr* and simple expressions *Sexpr*. Every expression has a type, namely the domain to which its value belongs.

Simple expressions include variables, constants, application of primitive operators, and application of user defined functions. Moreover, object constructors and dissectors are simple expressions. The following is a list of simple expressions allowed in MELA. The metavariable S ranges over *Sexpr*.

[D | c]: Denotes the enumerated constant of type D;

[D | S_1,..., S_n]: Denotes an element of the cartesian product $D = D_1$ x...x D_n whose components are the values of S_1,..., S_n;

[D | lambda (D_1: V_1 ;...; D_n: V_n) . E]: Denotes an object of the functional domain $D = D_1$ x...x D_n -> D_{n+1}, where D_{n+1} is the type of the expression E;

[D | Error "string"]: Denotes the *bottom* element of domain D. The string has two uses: as a comment

in the semantic definitions, and as an error message that is issued by the interpreter whenever this expression is evaluated;

P -> S: S must be a simple expression whose type belongs to a cartesian product D, and P is a projector of D.

Equality is defined only on primitive and enumerated domains. Identifiers must be compared using the *eqide* operator. Functional application is done according to call by value[2], evaluating arguments from left to right. If the function to be applied is denoted by a functional variable, this is evaluated first, before the arguments. The intended order of evaluation may be relevant whenever modeling input/output streams in programming languages.

Structure expressions include a non-strict conditional and three kind of let expressions used to introduce local variables. The scope of the variables introduced in a let expression is static and will be specified case by case. The simplest form of let is:

let D_1: V_1 = S_1;...; D_n: V_n = S_n in E

that introduces the variables V_1,..., V_n with scope E, binding them to the values of the expressions S_1,..., S_n (evaluated in this order). The value of the let expression is the value of E. This let expression is strict in S_1,..., S_n.

The letrec expression is similar, but it is limited to bind variables V_1,..., V_n to functional values S_1,..., S_n. The letrec expression is used to define a set of mutually recursive functions, with scope E.

The last form of let is to be used to select the type of a value belonging to a domain defined as a disjoint union. The letcase expression:

letcase D: V = S in D_1 => E_1;...; D_n => E_n

has the value of one of the expressions E_1,..., E_n. V is a variable of type D, whose initial value is given by S. V has type D_1,..., D_n within E_1,..., E_n respectively. The type of the value of V is used to select one of the cases of the disjoint union D, whose addends are D_1 ,..., D_n. The value of the chosen expression is the value of the whole letcase expression. Letcase expression is strict in S. A typical use of letcase is to discriminate the different cases of a syntactic domain.

[2]The choice of call-by-value rather than call-by-name semantics does not affect the expressive power of MELA. As shown in [22], it is always possible to simulate one with the other.

We conclude this section by giving the semantics of our sample language T. In Figure 4-5 we use the standard clause notation. Then the same clauses are translated into the semantic functions of a MELA program (see Figure 4-6).

The semantics of T is straightforward. Conditional and loop statements behave like the skip statement when the guard has a non-boolean value. The output stream has been modeled as a list. An error occurs whenever an unbound variable is accessed.

The style of semantic equations in MELA is different and more awkward than the usual clause notation. The type of an object must always be explicitly indicated, even when it may be inferred from the context. This choice has been taken only to simplify the automatic generation of the interpreter. We claim that this is not a real hindrance since it could be avoided through modification of the ALOE editor. In this sense MELA can be viewed as an intermediate level of specification between the mathematical style of writing denotational semantics and the mechanized tool that generates the environment.

Sprog: *PROG* -> *O*
Scom: *COM* -> *S* -> *S*
Sexpr: *EXPR* -> *M* -> *V*

Sprog [[**prog** C]] = o
  where ⟨*m*, *o*⟩ = *Scom* [[C]]⟨ ⟩
  and ⟨ ⟩ denotes the initial empty state
Scom [[**skip**]]*s* = *s*
Scom [[C_1; C_2]]*s* = *Scom* [[C_2]] (*Scom* [[C_1]]*s*)
Scom [[**if** E **then** C **fi**]]*s* = ((*v* ∈ *Bool* ∧ *v*) ⟶ *Scom* [[C]]*s*, *s*)
  where *s* = ⟨*m*, *o*⟩ and *v* = *Sexpr* [[E]]*m*
Scom [[**while** E **do** C **od**]]*s* = *t*(*s*)
  where *t* = λ*s*'. ((*v* ∈ *Bool* ∧ *v*) ⟶ *t*(*Scom* [[C]]*s*'), *s*')
    where *s*' = ⟨*m*', *o*'⟩ and *v* = *Sexpr* [[E]]*m*'
Scom [[I := E]]*s* = ⟨*m* + [I ⟶ *Sexpr* [[E]]*m*], *o*⟩
  where *s* = ⟨*m*, *o*⟩
Scom [[**write** E]]*s* = ⟨*m*, ⟨*Sexpr* [[E]]*m*, *o*⟩⟩
  where *s* = ⟨*m*, *o*⟩
Sexpr [[**true**]]*m* = *true*
Sexpr [[**false**]]*m* = *false*
Sexpr [[N]]*m* = N
Sexpr [[I]]*m* = *m*[[I]]
Sexpr [[**neg** (E)]]*m* = ((*v* ∈ *Bool*) ⟶ ~*v*, -*v*)
  where *v* = *Sexpr* [[E]]*m*

Figure 4-5: Semantic equations of T

```
Sprog (PROG: p): OUT <=
 let STATE: s = [STATE| InitM ( ), InitO ( )] in
  Out -> Scom (Com -> p, s)

Scom (COM: c; STATE: s): STATE <=
 let MEM: m = Mem -> s;
   OUT: o = Out -> s in
  letcase COM: c = c in
   SKIP => s;
   SEQ  => Scom (SeqCom2 -> c, Scom (SeqCom1 -> c, s));
   COND => letcase VAL: v = Sexpr (CondExpr -> c, m) in
        INT => s;
        BOOL => if v
             then Scom (CondCom -> c, s)
             else s;
   LOOP => letrec CMD: t = [CMD| lambda (STATE: st) .
        letcase VAL: v = Sexpr (LoopExpr -> c, Mem -> st) in
            INT => st;
            BOOL => if v
                 then t (Scom (LoopCom -> c, st))
                 else st]
       in t (s);
   ASSIGN => [STATE| UpdateM (m, AssignIde -> c,
            Sexpr (AssignExpr -> c, m)), o];
   WRITE => [STATE| m,
         AppendO (Sexpr (WriteExpr -> c, m), o)]

Sexpr (EXPR: e; MEM: m): VAL <=
 letcase EXPR: e = e in
  BOOL => e;
  INT => e;
  IDE => LookupM (m, e);
  NEG => letcase VAL: v = Sexpr (NegExpr -> e, m) in
       BOOL => not v;
       INT => - v


InitM ( ): MEM <=
 [MEM| lambda (IDE: i) . [UNB| unbound]]

UpdateM (MEM: m; IDE: I; VAL: v): MEM <=
 [MEM| lambda (IDE: i) .
    if eqide (i, I)
     then v
     else LookupM (m, I)]

LookupM (MEM: m; IDE: i): VAL <=
 letcase UVAL: u = m (i) in
  BOOL => u;
  INT => u;
  UNB => [VAL| Error "unbound var"]

InitO ( ): OUT <=
 [NIL| nil]

AppendO (VAL: v; OUT: o): OUT <=
 [STREAM| v, o]
```

Figure 4-6: Semantic equations of T in MELA

5. REMOVAL OF HIGHER ORDER DEFINITIONS

The idea of removing higher order functional domains was originally presented in [3] to transform a metacircular interpreter into an equivalent first order program. The application of this technique has been proposed in recent work on compiler construction [23, 24] and in the optimization of tools derived from denotational semantics of programming languages [25, 26].

In this section we introduce a complete set of transformations that take a MELA program and produce an equivalent program where all functional domains have been removed. These transformations introduce new domains, which substitute the removed functional domains, and new auxiliary functions that are used in the transformed program to compute the removed lambda expressions. The transformations presented in this section extend and complete those given in [3, 27].

We explain the set of transformations using three examples of increasing complexity[3]. Then we show how these transformations can be applied to the definition of the sample language T.

First example. The purpose of this example is to introduce the basic idea on which the entire method for removing functional domains is based. Consider the simple program P of Figure 5-1 which contains a single functional domain A and a single object of this domain, a lambda expression L.

```
A = Int -> Int

f (Int: y): Int <=
  let A: g = [A| lambda (Int: x) . (x + y)]
    in g (y) + y
```

Figure 5-1: Program P

The first step of the transformation process (see Figure 5-2) removes the functional domain A and introduces a new domain AR which substitutes all the occurrences of A in program P. The new domain AR is a cartesian product which has a single component of the same type as y, the only global variable contained in L. The lambda expression L is replaced with a tuple belonging to AR, containing the value of y. Note that P' is not correct because of the improper use of g in the expression g (y).

```
AR = Int: ry

f (Int: y) <=
  let AR: g = [AR| y]
    in g (y) + y
```

Figure 5-2: Program P'

The second step of the transformation introduces a new function AW, the interpretation function of the

[3] The careful reader will notice that the functions used in the examples are all equivalent to the simple function ($\lambda x.3x$)

domain AR, and substitutes the ill-formed expression g (y) with AW (g, y). The definition of AW corresponds to the body of the lambda expression L_1 in which the occurrence of y has been replaced with (ry -> r).

The interpretation function AW has the property that the evaluation of AW (g, y) in P" (see Figure 5-3) is equivalent to the evaluation of g (y) in P. The proof that P and P" are semantically equivalent is based on the static scoping of variables in MELA and is out of the scope of this work.

```
AR: Int: ry

f (Int: y) <=
  let AR: g = [AR| y]
    in AW (g, y) + y

AW (AR: r; Int: x) <=
  (x + (ry -> r))
```

Figure 5-3: Program P"

After the second step, P" contains neither functional domains nor lambda expressions and it is close to a program of a first order programming language. At the end of this section we show how a transformed program can be effectively translated into an equivalent C program.

Second example. The above transformations can be extended to cope with other cases of functional domains and expressions. The first generalization consists of transforming a lambda expression which contains two or more global variables. In this case the transformation introduces a cartesian domain whose components are equal in number and type to the global variables of the transformed lambda expression.

Furthermore, if a lambda expression has two or more parameters, it is sufficient to augment the number of parameters of the auxiliary interpretation function. When a program contains two or more lambda expressions, which belong to different functional domains, it is sufficient to introduce as many auxiliary domains and interpretation functions as there are lambda expressions being transformed.

When a program contains two or more lambda expressions which belong to the same functional domain, the above transformation is no longer correct. The transformed program would contain two auxiliary interpretation functions for the same functional domain. It would be impossible to decide which of them must be used in the second step of the transformation, when ill-formed functional applications are transformed.

To solve this problem, the first step of the transformation introduces a new domain defined as the union of all the auxiliary domains already introduced to transform these lambda expressions. Moreover, all the corresponding interpretation functions are merged into a single one that selects the right interpretation code by simply looking at the domain of its first parameter.

As an example, consider program Q in Figure 5-4 which contains two lambda expressions L_1 and L_2 belonging to the functional domain A. L_1 contains the global variable y and L_2 contains the global variables z and y.

```
A = Int -> Int

f (Int: y): Int <=
  let Bool: z = true
   in let A: g1 = [A| lambda (Int: x) . (x + y)];
          A: g2 = [A| lambda (Int: x) . (if z then x else y)]
      in g1 (y) + g2 (y)
```

Figure 5-4: Program Q

The first step of the transformation produces the program Q', shown in Figure 5-5, in which g1 and g2 are improperly applied. The second step produces the program Q" (see Figure 5-6) in which the interpretation function AW is correctly applied to g1 and g2.

```
AR1 = Int: ry1
AR2 = Int: ry2 x Bool: rz2
AR = AR1 + AR2

f (Int: y): Int <=
  let Bool: z = true
   in let AR: g1 = [AR1| y];
          AR: g2 = [AR2| y, z]
      in g1 (y) + g2 (y)
```

Figure 5-5: Program Q'

```
AR1 = Int: ry1
AR2 = Int: ry2 x Bool: rz2
AR = AR1 + AR2

f (Int: y): Int <=
  let Bool: z = true
   in let AR: g1 = [AR1| y];
          AR: g2 = [AR2| y, z]
      in AW (g1, y) + AW (g2, y)

AW (AR: r; Int: x): Int <=
  letcase R: r = r in
    AR1 => x + (ry1 -> r);
    AR2 => if (rz2 -> r) then x else (ry2 -> r)
```

Figure 5-6: Program Q"

In this example both lambda expressions have a parameter with an identical name. This allows the transformation to define an auxiliary interpretation function whose second parameter has that name. In general, since parameters can have different names, the second step of the transformation will perform an explicit alpha-conversion on the body of every lambda expression.

Third example. In the previous examples we did not consider any program containing letrec expressions. This omission is not casual. As we show in the following transformation, letrec expressions deserve a special treatment since their semantics involve the notion of mutual recursive definitions.

Consider the program R shown in Figure 5-7, a slightly more complex program than the previous programs P and Q. Variables g1 and g2 are mutually defined inside a letrec expression. There is also another variable h, defined in terms of g1 and g2, whose type is the same of g1 and g2. However, h is not mutually defined with g1 and g2.

In this example, the transformation considers g1 and g2 as global variables inside the definition of g1 and g2, and vice versa. It produces a transformed program containing the following definitions of g1 and g2:

```
letrec AR: g1 = [AR1| y, g1, g2];
    AR: g2 = [AR2| y, z, g1, g2]
 in ...
```

where g1 and g2 are bound to a tuple which contains the (still undefined) values of g1 and g2. The only possible meaning of these expressions is the undefined value. This situation is due to the semantics of the letrec expression. The values of g1 and g2 are given in terms of the fix-point of their definitions.

```
A: Int -> Int

f(Int: y): Int <=
  let Bool: z = true
   in letrec A: g1 = [A| lambda (Int: x) . if x = y
                          then x + y
                          else g1 (x) + g2 (y)];
         A: g2 = [A| lambda (Int: x) . if z
                          then x
                          else g1 (x) + g2 (y)]
      in let A: h = [A| lambda (Int: x) . g1 (x) + g2 (y)]
         in h (y)
```

Figure 5-7: Program R

To avoid this situation, the transformation does not consider g1 and g2 as global variables inside their definitions. However, the transformation substitutes the definition of g1 and g2 with a tuple which contains the value of the two global variables y and z. This tuple is called the global state of the definitions introduced in the letrec expression.

The first step of the transformation produces the program R' shown in Figure 5-8 in which, as usual, functional applications are ill-formed. The second step of the transformation would introduce the following interpretation function AW:

```
AW (AR: r; Int: x): Int <=
 letcase AR: r = r in
 AR1 => if x = (ry -> ar1 -> r)
        then x + (ry -> ar1 -> r)
        else AW (g1, x) +
             AW (g2, (ry -> ar1 -> r));
 AR2 => if (rz -> ar2 -> r)
        then x
        else AW (g1, x) +
             AW (g2, (ry -> ar2 -> r));
 AR3 => AW (rg1 -> r, ry3 -> r) +
        AW (rg2 -> r, ry3 -> r)
```

where the two variables g1 and g2 are unknown. To avoid the use of unknown variables, arising from letrec expressions, we have changed the second step of the transformation.

Consider the intended semantics of the expression AW (g1, x) inside the AR1 case of AW. It requires the application of the definition of g1 to the variable x. This can be achieved by transforming variable g1 into an object that causes the choice of the case AR1 in the interpretation function AW. The object must be a tuple belonging to AR whose value is another tuple of type AR1. Doing so, the interpretation function will select the case AR1 and will properly apply the body of g1 to y.

In order to evaluate the body of g1 within the right scope, the transformation must build a tuple of type AR1 which contains the values of the global variables of g1. However, this exactly corresponds to the state used for the evaluation of AW (g1, y), because it is the global state of the letrec expression. The actual value of the global state can be used for building the tuple of type AR1.

```
ARG = Int: ry x Bool: rz
AR1 = ARG: ar1
AR2 = ARG: ar2
AR3 = AR: rg1 x AR: rg2 x Int: ry3
AR = AR1 + AR2 + AR3

f(Int: y; Bool: z): Int <=
  let Bool: z = true
   in let AR: g1 = [AR1| [ARG| y, z]];
         AR: g2 = [AR2| [ARG| y, z]]
      in let AR: h = [AR3| g1, g2, y]
         in h (y)
```

Figure 5-8: Program R'

Figure 5-9 shows the final version R" of the program R, obtained by applying the second step of the transformation, properly modified. The third case AR3 inside AW has not been influenced by the modification of the second step. In fact, in the definition of h, g1 and g2 are treated as real global variables.

```
ARG = Int: ry x Bool: rz
AR1 = ARG: ar1
AR2 = ARG: ar2
AR3 = AR: rg1 x AR: rg2 x Int: ry3
AR = AR1 + AR2 + AR3

f(Int: y; Bool: z): Int <=
  let Bool: z = true
   in let AR: g1 = [AR1| [ARG| y, z]];
        AR: g2 = [AR2| [ARG| y, z]]
      in let AR: h = [AR3| g1, g2, y]
         in AW (h, y)

AW (AR: r; Int: x): Int <=
  letcase AR: r = r in
   AR1 => if x = (ry -> ar1 -> r)
        then x + (ry -> ar1 -> r)
        else AW (r, x) + AW ([AR2| ar1 -> r], (rgy -> ar1 -> r)):
   AR2 => if (rz -> ar2 -> r)
        then x
        else AW ([AR1| ar2 -> r], x) + AW (r, (ry -> ar2 -> r)):
   AR3 => AW (rg1 -> r, ry3 -> r) + AW (rg2 -> r, ry3 -> r)
```

Figure 5-9: Program R"

We have a last comment to the treatment of the letrec expression. In the last example we have assigned a lambda expression to variables g1 and g2. This is not a general case because the syntax of MELA allows the use of a generic expression in the definition of a letrec variable. However, since letrec variables are restricted to belong to functional domains, only a few expressions can be used to define their values. For instance, integer and boolean expressions are not allowed.

Nevertheless, it is still possible to define the value of a letrec variable without using a lambda expression. This case is treated by transforming the expression E, different from a lambda expression, into the equivalent:

$$[D|\ \text{lambda}\ (D_1: V_1; \ldots; D_n: V_n).E\ (V_1, \ldots, V_n)]$$

where the type of E is $D = D_1 \times \ldots \times D_n \rightarrow D_{n+1}$ and $V_1, \ldots, V_n$ are not free variables in E.

Let us now turn to the definition of the sample language T. The first step of the transformation removes the functional domains MEM and CMD, and introduces the auxiliary domains MEMR and CMDR. The second step introduces the interpretation functions MEMW and CMDW, as shown in Figure 5-10. Finally in Figure 5-11 we show the functions Scom, InitM, UpdateM, and LookupM after the transformation process.

```
MEMR1 = {emptym}
MEMR2 = IDE: rl x VAL: rv x MEMR: rm
MEMR = MEMR1 + MEMR2
CMDR = COM: rc

MEMW (MEMR: r; IDE: i): UVAL <=
   letcase MEMR: r = r in
    MEMR1 => [UNB| unbound];
    MEMR2 => if eqide (i, rl -> r)
          then (rv -> r)
          else LookupM (rm -> r, rl -> r)

CMDW (CMDR: r; STATE: st): STATE <=
   letcase VAL: v = Sexpr (LoopExpr -> rc -> r, Mem -> st) in
    INT => st;
    BOOL => if v
          then CMDW (r, Scom (LoopCom -> rc -> r, st))
          else st
```

Figure 5-10: Domains and interpretation functions for T

```
Scom (COM: c; STATE: s): STATE <=
 let MEMR: m = Mem -> s;
   OUT: o = Out -> s in
  letcase COM: c = c in
  SKIP => s;
  SEQ => Scom (SeqCom2 -> c, Scom (SeqCom1 -> c, s));
  COND => letcase VAL: v = Sexpr (CondExpr -> c, m) in
        INT => s;
        BOOL => if v
              then Scom (CondCom -> c, s)
              else s;
  LOOP => let CMDR: t = [CMDR| c]
        in CMDW (t, s);
  ASSIGN => [STATE| UpdateM (m, AssignIde -> c,
              Sexpr (AssignExpr -> c, m)), o];
  WRITE => [STATE| m, Output (Sexpr (WriteExpr -> c, m), o)]

InitM ( ): MEMR <=
 [MEMR1| emptym]

UpdateM (MEMR: m; IDE: l; VAL: v): MEMR <=
 [MEMR2| l, v, m]

LookupM (MEMR: m; IDE: i): VAL <=
 letcase UVAL: u = MEMW (m, i) in
  BOOL => u;
  INT => u;
  UNB => [VAL| Error "unbound var"]
```

Figure 5-11: Semantic equations in T involved in the transformation

6. GENERATION OF THE ENVIRONMENT WITH THE INTERPRETER

The generation of the interpreter for a programming language L and the integration of the interpreter in a GANDALF environment has been outlined in Section 3. In the previous section we have described the first step of the generation process, *i.e.* the removal of higher order functions. The other two steps, that we present in this section, are the derivation of abstract

syntax from the domain definitions, and the generation of the interpreter from the semantic equations.

6.1. Domain Implementation

A MELA program that does not contain functional domains can be easily translated into an equivalent C program. This corresponds to giving an operational semantics to this subset of MELA programs. In this section we describe how syntactic and semantics domains are mapped onto an equivalent abstract syntax definition given in the ALOEGEN notation.

The translation of the domains defined in the syntactic part of a MELA program is performed by the ALOE for MELA by means of an unparsing scheme which performs the following actions:

- $D = \{C\}$: Enumerated domain D is translated into a terminal static node D. Static nodes are used in ALOEGEN to implement syntactic components that correspond to single keywords, like the SKIP statement in the language T.
- $D = D_1 + ... + D_n$: Disjoint union D is translated into a class D. The elements of D are $D_1 ... D_n$. ALOEGEN requires that the elements of a class be non-terminal or terminals. Recall that in MELA a disjoint union cannot be defined in terms of other unions. This restriction insures the translation's correctness.
- $D = D_1: P_1 x ... x D_n: P_n$: Cartesian product D is translated into a non terminal D whose components are the classes $D'_1 ... D'_n$. Given a component D_i, there are two possible cases: either D_i is defined as a disjoint union, or it is defined as another cartesian product, an enumerated domain, or a primitive domain. In the former case D'_i corresponds to D_i; in the latter case the translation introduces a new class D'_i whose name is defined as the concatenation of P_i and D_i and which is defined as a class containing a single element D_i. Again this translation produces an abstract syntax consistent with the ALOEGEN conventions.

Figure 6-1 shows the abstract syntax corresponding to the syntactic domains of the language T.

Semantic domains are translated in almost the same way, with the only exception of enumerated domains which are translated into string terminals, *i.e.* terminals whose associated value is a string. This solution allows the successive translation of a simple expression formed by an enumerated constant as a string terminal containing the name of the constant itself. Figure 6-2 shows the abstract syntax corresponding to the semantic domains of the language T.

Non-terminal definitions

```
PROG     = COM
SEQ      = COM COM
COND     = EXPR COM
LOOP     = EXPR COM
ASSIGN   = AssignIdeIDE EXPR
WRITE    = EXPR
NEG      = EXPR
```

Terminal definitions

```
SKIP     = {static}
```

Class definitions

```
COM      = SKIP SEQ COND
           LOOP ASSIGN WRITE
EXPR     = BOOL INT IDE NEG
AssignIdeIDE = IDE
```

Figure 6-1: Abstract syntax of syntactic domains of T

The abstract syntax obtained from syntactic domains is used to generate an ALOE editor for the language, while the abstract syntax obtained from semantic domains is used to provide auxiliary semantic structures to the editor. The final user of the environment is never aware of the existence of these semantic structures that are used exclusively by the interpreter. The advantage of this solution consists of the use of the facilities provided by the ALOE action routine model for maintaining auxiliary semantic structures.

Non-terminal definitions

```
STREAM = VAL OUT
STATE  = MEMR OUT
MEMR2  = rlIDE VAL MEMR
CMDR   = COM
```

Terminal definitions

```
UNB    = {string}
NIL    = {string}
MEMR1  = {string}
```

Class definitions

```
VAL    = INT BOOL
UVAL   = INT BOOL UNB
OUT    = NIL STREAM
MEMR   = MEMR1 MEMR2
rlIDE  = IDE
```

Figure 6-2: Abstract syntax of semantic domains of T

6.2. Function Implementation

The final step in the interpreter generation process consists of the generation of a C program from the semantic equations obtained after the removal of higher order functions. The generation is imple-

mented as a mapping from the syntactic categories of MELA into a C program in textual form. The generation is completely context-free and can be seen as a macro expansion. It has been implemented in the ALOE editor for MELA using a special unparsing scheme that produces a C program from the abstract syntax tree of a MELA program.

The definition of the transformation of the equations is given in the Appendix II. The translation functions:

D : Dom -> Instr *F : Fun -> Instr*
E : Expr -> Instr *S : Sexpr -> Instr*

are mappings from MELA syntactic categories into strings. The right hand sides of the clauses defining each mapping has to be understood as a string concatenation. For example, the clause

$$E[\![\text{P -> S}]\!] = \text{getson}(E[\![\text{S}]\!], \text{P})$$

has the following meaning: the expression P -> S is transformed into the string made up of the string "getson (" concatenated with the string resulting from the transformation of S, concatenated with the string ",", the value of the syntactic metavariable P, and ")".

The final C program is obtained by appending the result of these transformations to some book-keeping code: a sequence of definitions obtained from ALOEGEN, in which domain identifiers are associated with the numbers internally used by ALOEGEN, and the declaration of all the *bottom functions* used in the semantic functions.

The resulting C program is compiled, and then it is added as an extended command to the purely syntactic editor previously generated using the abstract syntax produced in the domain implementation. The final environment includes the editor and the interpreter. Appendix III shows the interpreter for the language T.

6.3. Further Improvements

The typical drawback of the application of a transformational technique is the inefficiency of the resulting programs. We have found this problem applying the generation process to a set of programming language definitions. However, we believe that the efficiency of the resulting interpreter can be improved by using other standard transformations, such as tail-recursion removal and constant propagation.

Other improvements can be achieved using different implementations for semantic domains. For instance they could be implemented defining an explicit C structure for every semantic domain, mapping cartesian products onto structures and disjoint unions onto unions, and defining at the same time their allocators. This would lead, however, to duplicated code in the resulting system.

Some savings in memory space may be obtained by exploiting the built-in facilities of ALOEGEN to deal with syntactic categories that are in fact lists of items. This improvement requires the introduction of a new domain constructor in MELA, to be seen semantically as shorthand for the explicit list definition that must be used in MELA as it stands. See, for instance, the definition of the output stream in the semantics of T. The translation into the abstract syntax might then easily map these domain definitions onto ALOE lists.

Moreover, provided that the definition of the semantic domains is given in an *abstract data type* style, implementations different from that provided by the use of ALOE auxiliary semantic structures may be used. For instance, since the output domain is used only by the auxiliary function output, the implementation of this function may be changed so that it behaves like the identity over OUT, but it has the side-effect of printing its first argument.

Similar improvements may be obtained by special implementation of the domains modeling memories, environments, etc.. The trade-off between the gain in efficiency and the cost of implementation is related to the goals of the required environment in the scale from rapid prototyping to almost *production* environments. On the other side, this approach may lead to the creation of reusable libraries of *standard* semantic domain implementations.

Although the approach has been presented for the interpreter generation, it can be extended to the generation of other tools such as type-checkers, scopers and so on, exploiting non-standard semantics of the language. However, the lack of interactivity and incrementality in the derived tools makes their use unsuitable in a GANDALF programming environment. Nevertheless, the generation technique can be used to derive the specification of static semantic tools whose final implementation, with respect to incrementality and interactivity, will be obtained using other generation techniques [18].

Finally, with some minor modifications to the transformations into C described in the Appendix II, the method may be applied to the definition of MELA itself. This will lead to the generation of a MELA interpreter, which, although inefficient, could make available all the run-time structures to an ALOE, providing the basis for a user-friendly debugger, smoothly integrated in the environment.

7. CONCLUSIONS

In this paper we show how a GANDALF environment can be exploited to automatically derive the interpreter of a programming language, starting from a formal definition of the language and applying a set of transformations. The resulting system is itself a GANDALF environment.

Still open problems are the improvement of efficiency of the generated tools and their full integration in the resulting environment. The solution of these problems will represent a further step towards the generation of semantically based programming environments.

ACKNOWLEDGEMENTS

M. Venturini, R. Carino, and C. Costanzo implemented preliminary versions of the system reported in this paper.

REFERENCES

1. Bauer, F. L., Broy, M., Dosh, W., Gnatz, R., Krieg-Brueckner, Laut, A., Matzner, T., Moeller, B., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., and Woessner, H., Report on a Wide Spectrum Language for Program Specification and Development, Tech. report, Institute fur Informatik, Technische Universitat, Munchen, 1981.
2. Darlington, J., An Experimental Program Transformation and Synthesis System, *Artificial Intelligence*, Vol. 16, 1981.
3. Reynolds, J.C, Definitional Interpreters for Higher-Order Programming Languages, *ACM National Conference*, 1972, pp. 717-740.
4. Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
5. Johnson, G. F., and Fischer, C. N., Non-syntactic Attribute Flow in Language Based Editors, *9th Annual ACM Symposium on Principles of Programming Languages*, ACM, January 1982, pp. 185-195.
6. Reps, T., and Teitelbaum, T., The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *Comm. ACM*, Vol. 24, No. 9, Sept. 1981, pp. 563-573.
7. Reps, T., Teitelbaum, T., and Demers, A., Incremental Context-Dependent Analysis for Language-Based Editors, *ACM Trans. on Prog. Lang. and Sys.*, Vol. 5, No. 3, July 1983, pp. 449-477.
8. Reiss, S. P., PECAN: Program Development Systems that Support Multiple Views, *7th International Conference on Software Engineering*, March 1984.
9. Reiss, S. P., Generation of Compiler Symbol Processing Mechanism from Specifications, *ACM Trans. on Progr. Lang. and Sys.*, Vol. 5, No. 2, April 1983, pp. 127-163.
10. Ambriola, V., Kaiser, G. E., and Ellison, R., An Action Routine Model for ALOE, Tech. report CMU-CS-84-156, Carnegie-Mellon University Department of Computer Science, August 1984.
11. Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., Programming Environments Based on Structured Editors: the MENTOR Experience, Tech. report 26, INRIA, May 1980.
12. Kaiser, G. E., and Feiler, P. H., Generation of Language-Oriented Editors, *Programmierumgebungen und Compiler*, German Chapter of the ACM, April 1984.
13. Barbuti, R., Bellia, M., Degano, P., Levi, G., and Martelli, A., Programming Environments: Deriving Language Dependent Tools from Structured Denotational Semantics, *International Comp. Symposium on Application Systems Developments, Nurnberg*, 1983.
14. Barbuti, R., and Martelli, A., A Structured Approach to Static Semantics Correctness, To appear in *Science of Computer Programming*.
15. Cousot, P., and Cousot, R., Static Determination of Dynamic Properties of Programs, *International Symposium on Programming, Paris*, 1976.
16. Donzeau-Gouge, V., Utilization de la Semantic Denotationelle pour la Description d'interpretations Non-Standard: application a la Validation et a l'optimization des Programmes, *International Symposium on Programming, Paris*, 1978.
17. Jager, M., Bahlke, R., Wenhalp, W., Hunkel, M., Letschert, T., and Snelting, G., PSG - Programming System Generator, *Programmierumgebungen und Compiler*, German Chapter of the ACM, April 1984.
18. Kaiser, G. E., *Semantics of Structure Editing Environments*, PhD dissertation, Carnegie-Mellon University, 1985, In progress.
19. Stoy, J., Semantic Models, *Theoretical Foundations of Program Methodology*, D. Reidel, 1981.
20. Gordon, M.J.C., *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.
21. Stoy, J., *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT press, 1977.
22. Plotkin, G. D., Call-by-name, Call-by-value and the Lambda-Calculus, *Theoretical Computer Science*, Vol. 1, 1975, pp. 125-159.
23. Ganzinger, H., Transforming Denotational Semantics into Practical Attribute Grammars, *Lect. Notes on Comp. Sc. 94*, Springer-Verlag, 1980.
24. Schmidt, D., State Transformation Machines for Lambda-Calculus Expressions, *Lect. Notes on Comp. Sc. 94*, Springer-Verlag, 1980.
25. Barbuti, R., Bellia, M., Degano, P., Levi, G., Dameri, E., Simonelli, C., and Martelli, A., Programming Environment Generation Based on Denotational Semantics, *Theory and Practice of Software Technology, D.Ferrari, M.Bolognani and*

J.Goguen eds., North-Holland, 1983.
26. Bjorner, D., Programming Languages: Formal Development of Interpreters and Compilers, *Int. Comp. Symposium, Morlet E. and Ribbens D. eds*, North-Holland, 1977.
27. Lucchesi, L., Formal Definition, Definitional Interpreters and Tools1982, Dipartimento di Informatica, University of Pisa, Master Thesis. In Italian.

APPENDIX 1. BNF DEFINITION OF MELA

This is the definition of MELA in BNF style. It is organized in four parts. The first part contains the description of syntactic and semantic domains. The second part describes functions. The third and fourth part describe structured and simple expressions. We assume that the syntactic categories of the form *<...name>* belong to the domain Ide of identifiers.

Syntactic and Semantic Domains

<prog> ::= *<syntpart>* | *<sempart>* | *<funpart>*

<syntpart> ::= **synt dom** *<domlist>*

<sempart> ::= **sem dom** *<domlist>*

<domlist> ::= *<dom>* | *<dom>* *<domlist>*

<dom> ::= *<enumdom>* | *<uniondom>* |
<cartdom> | *<fundom>*

<enumdom> ::= *<domname>* = { *<constlist>* }

<constlist> ::= *<constname>* | *<constname>* , *<constlist>*

<uniondom> ::= *<domname>* = *<unionlist>*

<unionlist> ::= *<domname>* | *<domname>* + *<unionlist>*

<cartdom> ::= *<domname>* = *<cartlist>*

<cartlist> ::= *<domname>* : *<projname>* |
<domname> : *<projname>* x *<cartlist>*

<fundom> ::= *<domname>* = *<funlist>* -> *<domname>*

<funlist> ::= *<domname>* | *<domname>* x *<funlist>*

Functions

<funpart> ::= **functions** *<funclist>*

<funclist> ::= *<fun>* | *<fun>* *<funclist>*

<fun> ::= *<funname>* *<parlist>* : *<domname>* <= *<expr>*

<parlist> ::= () | (*<nparlist>*)

<nparlist> ::= *<domname>* : *<varname>* |
<domname> : *<varname>* ; *<nparlist>*

Structure Expressions

<expr> ::= *<sexpr>* |
<cond> |
<let> |
<letrec> |
<letcase>

<cond> ::= **if** *<sexpr>*
then *<expr>*
else *<expr>*

<let> ::= **let** *<letlist>* **in** *<expr>*

<letrec> ::= **letrec** *<letlist>* **in** *<expr>*

<letcase> ::= **letcase** *<letvar>* **in** *<caselist>*

<letlist> ::= *<letvar>* |
<letvar> ; *<letlist>*

<letvar> ::= *<domname>* : *<varname>* = *<sexpr>*

<caselist> ::= *<domname>* => *<expr>* |
<domname> => *<expr>* ; *<caselist>*

Simple Expressions

<sexpr> ::= **true** | **false** | (*<sexpr>*) |
<number> |
<varname> |
<op1> *<sexpr>* |
<sexpr> *<op2>* *<sexpr>* |
eqide (*<sexpr>* , *<sexpr>*) |
<projname> -> *<sexpr>* |
[*<domname>* | **Error** " *<string>* "] |
[*<domname>* | *<constname>*] |
[*<domname>* | *<sexprlist>*] |
[*<domname>* | **lambda** *<parlist>* . *<expr>*]
<varname> *<sexprlist>* |
<funname> *<sexprlist>*

<sexprlist> ::= () |
(*<nsexprlist>*)

<nsexprlist> ::= *<sexpr>* |
<sexpr> , *<nsexprlist>*

<op1> ::= **not** | -

<op2> ::= **and** | **or** | + | * | -

APPENDIX 2. OPERATIONAL SEMANTICS OF MELA

In this appendix we give the operational semantics of MELA in terms of textual transformations that generate a C program. The execution of this program corresponds to the execution of the MELA transformed program.

The transformations described in this appendix are completely context-independent. This property is exploited by the ALOEGEN editor of MELA that generates a C program simply using an unparsing scheme which corresponds to the below transformations. Another way of looking at the process of

generating the C code is to consider these transformations as macros, and the whole process as a code generation via macro expansion.

Simple Expression Translation

S: *Sexpr* -> *Instr*

$S[\![\text{true}]\!] = \texttt{tt}$

$S[\![\text{false}]\!] = \texttt{ff}$

$S[\![\text{number}]\!] = \texttt{mk(iINT, number)}$

$S[\![V]\!] = V$

$S[\![\text{not } S]\!] = \texttt{mk(iBOOL, !val(}S[\![S]\!]\texttt{))}$

$S[\![-S]\!] = \texttt{mk(iINT, -val(}S[\![S]\!]\texttt{))}$

$S[\![S_1 \text{ and } S_2]\!] = \texttt{mk(iBOOL, val(}S[\![S_1]\!]\texttt{) \&\& val(}S[\![S_2]\!]\texttt{))}$

$S[\![S_1 \text{ or } S_2]\!] = \texttt{mk(iBOOL, val(}S[\![S_1]\!]\texttt{) || val(}S[\![S_2]\!]\texttt{))}$

$S[\![S_1 + S_2]\!] = \texttt{mk(iINT, val(}S[\![S_1]\!]\texttt{) + val(}S[\![S_2]\!]\texttt{))}$

$S[\![S_1 - S_2]\!] = \texttt{mk(iINT, val(}S[\![S_1]\!]\texttt{) - val(}S[\![S_2]\!]\texttt{))}$

$S[\![S_1 * S_2]\!] = \texttt{mk(iINT, val(}S[\![S_1]\!]\texttt{) * val(}S[\![S_2]\!]\texttt{))}$

$S[\![S_1 = S_2]\!] = \texttt{mk(iBOOL, val(}S[\![S_1]\!]\texttt{) == val(}S[\![S_2]\!]\texttt{))}$

$S[\![\text{eqide}(S_1, S_2)]\!] = \texttt{mk(iBOOL, eqide(val(}S[\![S_1]\!]\texttt{), val(}S[\![S_2]\!]\texttt{)))}$

$S[\![P \text{->} S]\!] = \texttt{getson(}S[\![S]\!]\texttt{, P)}$

$S[\![[D|\text{ Error "string"}]]\!] = \texttt{DSysAbort("string")}$

$S[\![[D|\,C]]\!] = \texttt{mk(iD, "C")}$

$S[\![[D|\,S_1, ..., S_n]]\!] = \texttt{mk(iD, }S[\![S_1]\!], ..., S[\![S_n]\!]\texttt{)}$

$S[\![F(S_1, ..., S_n)]\!] = \texttt{F(}S[\![S_1]\!], ..., S[\![S_n]\!]\texttt{)}$

Domain Translation

D: *Dom* -> *Instr*

$D[\![D = \{C_1, ..., C_n\}]\!] = \texttt{typedef tnode *D;}$

$D[\![D = D_1: P_1 \text{ x...x } D_n: P_n]\!] =$

```
#define p1   0
  ...
#define pn   n - 1
typedef tnode *D;
```

$D[\![D = D_1 + ... + D_n]\!] = \texttt{typedef tnode *D;}$

Function Translation

F: *Fun* -> *Instr*

$F[\![F(D_1: V_1; ...; D_n: V_n): D <= E]\!] =$

```
D F(V1 ..... Vn)
D1 V1;
 ...
Dn Vn;
{E[[E]]}
```

Structured Expression Translation

E: *Expr* -> *Instr*

$E[\![S]\!] = \texttt{return (}S[\![S]\!]\texttt{);}$

$E[\![\text{if S then } E_1 \text{ else } E_2]\!] =$

```
if val(S[[S]])
 {E[[E1]]}
  else E[[E2]];
```

$E[\![\text{let } D_1: V_1 = S_1; ...; D_n: V_n = S_n \text{ in E}]\!] =$

```
{D1 $V1;
  ...
 Dn $Vn;
 $V1 = S[[S1]];
  ...
 $Vn = S[[Sn]];
 {D1 V1;
   ...
  Dn Vn;
  V1 = $V1;
   ...
  Vn = $Vn;
  E[[E]];};
```

$E[\![\text{letcase D: V = S in } D_1 \Rightarrow E_1 ; \ldots ; D_n \Rightarrow E_n]\!] =$

```
{D $V;
 $V = S[[S]];
 {D V;
  V = $V
  switch (op (V) {
    case iD1 : E[[E1]] break;
    ...
    case iDn : E[[En]] break; }; }; };
```

APPENDIX 3. AN EXAMPLE OF TRANSLATION INTO C

In this appendix we show the complete interpreter for the language T generated by the transformations.

```
/* Projector Definitions */
#define Com 0
#define SeqCom1 0
#define SeqCom2 1
#define CondExpr 0
#define CondCom 1
#define LoopExpr 0
#define LoopCom 1
#define AssignIde 0
#define AssignExpr 1
#define WriteExpr 0
#define NegExpr 0
#define Last 0
#define Prev 1
#define rl 0
#define rv 1
#define rm 2
#define rc 0

/* Domain Name Definitions */
typedef tnode *PROG
typedef tnode *COM
typedef tnode *SKIP
typedef tnode *SEQ
typedef tnode *COND
typedef tnode *LOOP
typedef tnode *ASSIGN
typedef tnode *WRITE
typedef tnode *EXPR
typedef tnode *NEG
typedef tnode *VAL
typedef tnode *UNB
typedef tnode *UVAL
typedef tnode *NIL
typedef tnode *OUT
typedef tnode *STREAM
typedef tnode *STATE
typedef tnode *MEMR1
typedef tnode *MEMR2
typedef tnode *MEMR
typedef tnode *CMDR

/* Semantic Functions */
OUT Sprog (p)
PROG p;
{STATE s;
 s = mk (iSTATE, InitM (), InitO ());
 return(getson(Scom(getson(p, Com), s), Out));
}

STATE Scom (c, s)
COM c;
STATE s;
{MEMR $m;
 OUT $o;
 $m = getson (s, Mem);
 $o = getson (s, Out);
 {MEMR m;
  OUT o;
  m = $m;
  o = $o;
  {COM $c;
   $c = c;
   {COM c;
    c = $c;
    switch (op ($c)){
    case iSKIP:
      return (s);
      break;
    case iSEQ:
      return (Scom (getson (c, SeqCom2),
                    Scom (getson (c, SeqCom1),
                          s)));
      break;
    case iCOND:
      {VAL $v;
       $v = Sexpr (getson (c, CondExpr), m);
       {VAL v;
        v = $v
        switch (op (v)){
        case iINT:
          return (s);
          break;
        case iBOOL:
          if (val (v))
            return(Scom(getson(c,CondCom),s));
          else return (s);
          break;
        };
       };
      };
      break;
    case iLOOP:
      {CMDR $t;
       $t = mk (iCMDR, c);
       {CMDR t;
        t = $t;
        return (CMDW (t, s));
       };
      };
      break;
    case iASSIGN:
      return(
        mk(iSTATE,
           UpdateM(m,getson(c,AssignIde),
                   Sexpr(getson(c,AssignExpr),
                         m)),
           o));
      break;
    case iWRITE:
      return(
         mk (iSTATE,m,
             Output(Sexpr(getson(c,WriteExpr),
                          m),
                    o)));
      break;
    };
   };
  };
 };
}
```

```
VAL Sexpr (e, m)
EXPR e;
MEMR m;
{EXPR $e;
 $e = e;
 {EXPR e;
  e = $e;
  switch (op (e))
   {case iBOOL:
      return (e);
      break;
    case iINT:
      return (e);
      break;
    case iIDE:
      return (LookupM (M, e));
      break;
    case iNEG:
      {VAL $v;
       $v = Sexpr (getson (e, NegExpr), m);
       {VAL v;
        v = $v;
        switch (op (v))
         {case iBOOL:
            return (mk (iBOOL, !val (v)));
            break;
          case iINT:
            return (mk (iINT, -val (v)));
            break;
         };
       };
      };
      break;
   };
 };
}

MEMR InitM ()
{return (mk (iMEMR1, "emptym"));
}

MEMR UpdateM (m, l, v)
MEMR m;
IDE l;
VAL v;
{return (mk (iMEMR2, l, v, m));
}

VAL LookupM (m, i)
MEMR m;
IDE i;
{
 {UVAL $u;
  $u = MEMW (m, i);
  {UVAL u;
   u = $u;
   switch (op (u))
    {case iBOOL:
       return (u);
       break;
     case iINT:
       return (u);
       break;
     case iUNB:
       return (VALSysAbort ("unbound var"));
       break;
    };
  };
 };
}

OUT InitO ()
{return (mk (iNIL, "nil"));
}

OUT AppendO (v, o)
VAL v;
OUT o;
{return (mk (iSTREAM, v, o));
}

/* Auxiliary Function Definitions */

UVAL MEMW (r, i)
MEMR r;
IDE i;
{
 {MEMR $r:
  r = $r;
  {MEMR r;
   r = $r;
   switch (op (r))
    {case iMEMR1:
       return (mk (iUNB, "unbound"));
       break;
     case iMEMR2:
       if (val (eqide (i, getson (r, rl))))
         {return (getson (r, rv));}
       else
         {return(LookupM(getson(r,rm),
                         getson(r,rl)));}
       break;
    };
  };
 };
}

STATE CMDW (r, st)
CMDR r;
STATE st;
{
 {VAL $v;
  $v = Sexpr(getson(getson(r,rc),LoopExpr),
             getson(st,Mem));
  {VAL v;
   v = $v;
   switch (op (v))
    {case iINT:
       return (st);
       break;
     case iBOOL:
       if (val (v))
         {return(CMDW(r,
                      getson(r,rc),
                      LoopCom),st);}
       else return (st);
       break;
    };
  };
 };
}

/* Bottom Function Definitions */

VAL VALSysAbort (s)
char [] s;
{SysAbort (s);
}
```