Opal: Design And Implementation of an Algebraic Programming Language[⋆]

Klaus Didrich[1], Andreas Fett[2], Carola Gerke[1], Wolfgang Grieskamp[1], Peter Pepper[1]

[1] Technische Universität Berlin, Fachbereich Informatik, Institut für angewandte Informatik, Franklinstr. 28/29, 10587 Berlin
[2] now at Daimler-Benz AG, Forschung Systemtechnik, Alt-Moabit 91b, 10559 Berlin

1 Introduction

The algebraic programming language Opal has been designed as a testbed for experiments with the specification and development of functional programs. The language shall in particular foster the (formal) development of production-quality software that is written in a purely functional style. As a consequence, Opal molds concepts from *Algebraic Specification* and *Functional Programming* into a unified framework.

The amalgamation of these two concepts — which are both based on a sound theoretical foundation — constitutes an ideal setting for the construction of formally verified "safe" software. Moreover, the high level of abstraction and the conceptual clarity of the functional style can increase the productivity of software development significantly.

Unfortunately, the acceptance of the functional style suffers from the widespread prejudice that its advantages are paid for by a high penalty in runtime and storage consumption. Yet, this is not true.

Due to their semantic elegance and clarity, functional programs are amenable to a wide variety of powerful optimizations, such that the programmer need not be concerned with low-level considerations about machine pecularities. This way, she can concentrate on finding good algorithmic solutions instead.

The potential for automatic optimizations is highly increased, when functional programs are combined with algebraic specifications. Then, algebraic properties of the functional programs can be used to guide optimizations which go far beyond traditional optimization techniques.

In this paper we illustrate some principles of algebraic programming. Moreover, we introduce the language Opal, sketch its compilation strategies, and point out some challenges for further research in the area of algebraic programming languages.

[⋆] This work was partially sponsored by the German Ministry of Research and Technology (BMFT) as part of the project "KORSO – Korrekte Software".

2 The Functional Core of OPAL

The core of OPAL is a strongly typed, higher-order, *strict* functional language, which belongs to the tradition of HOPE and ML ([10, 7]). Due to our orientation towards algebra-based programming environments, the core language already has a distinctive algebraic flavour in the tradition of languages like CIP-L, ACT ONE, OBJ, and others ([1, 3, 2, 5]). This flavour shows up in the syntactical appearance of OPAL, in the preference of parameterization to polymorphism, and last but not least, in the semantics of OPAL. In the following, we first consider some examples, then point out the specialities of OPAL, and finally sketch the semantics.

2.1 Items

The basic syntactical parts of OPAL are *items*. The following OPAL text consists of four items, which declare and define the data type of sequences and the sequence concatenation:

```
DATA seq  == <>                        -- empty sequence
             ::(ft: data, rt: seq) -- first element and rest sequence
FUN ++ : seq ** seq -> seq     -- declaration of concatenation function

DEF (Ft :: Rt) ++ S == Ft :: (Rt ++ S)
DEF <>         ++ S == S
```

- The DATA item is in fact a shortcut for the following items:

```
SORT seq
FUN <>        :  seq                      -- constructors
FUN ::        :  data ** seq  ->  seq
FUN <>? ::?   :  seq          ->  bool   -- discriminators
FUN ft        :  seq          ->  data   -- selectors
FUN rt        :  seq          ->  seq
```

  It does not matter whether we add these declarations explicitly to the text or not since declarations can be repeated; collections of declarations collapse to sets.
- The DATA item does not only induce the above declarations, but also a *free algebraic type*, which ensures for example the following properties:

$$\forall ft_1, ft_2, rt_1, rt_2.\ ft_1 \not\equiv ft_2 \vee rt_1 \not\equiv rt_2 \Rightarrow ft_1 :: rt_1 \not\equiv ft_2 :: rt_2$$
$$\forall s.\ s \equiv \texttt{<>} \vee (\exists d, s'.\ s \equiv d :: s')$$

  These axioms, together with the generation principle (all elements of seq can be denotated by the constructors), allow for the pattern matching used in the left-hand sides of the definitions of the concatenation function ++. They also contain enough information to induce the definitions for the automatically declared functions.

2.2 Structures

In Opal, a program is a collection of "structures" which are connected to each other by import relations. Motivated by principles from software engineering, as well as by compilation aspects, we have split every structure into two parts:

structure = (visible) signature + implementation.

The signature part provides the *externally visible view* of the structure, and the implementation part provides the *hidden internal view*. That is, the signature is the syntactic interface of the software module, whereas the implementation constitutes its constructive realization.

An interface to a structure which implements edge-labeled directed graphs might look as shown in Table 1.

Table 1. A Signature of Edge-Labeled Graphs

```
SIGNATURE Graph[label]
  SORT label                          -- the parameter

  IMPORT Seq[node] ONLY seq           -- selective import
         Seq[edge] ONLY seq           -- only seq is visible here

  SORT graph edge node

  FUN empty  : graph                     -- empty graph

  FUN nodes  : graph -> seq[node]        -- nodes of graph
  FUN =      : node ** node -> bool      -- equal nodes
  FUN outs   : graph ** node -> seq[edge] -- outgoing edges
  FUN source                             -- source of edge
      destin : edge  -> node             -- destination of edge
  FUN label  : edge  -> label            -- label of edge

    -- and many more functions on graphs
```

Note that the structure `Graph` is *parameterized* by the sort `label`. On the basis of this interface, we may define a structure for the calculation of *minimal distances* in a directed edge-labeled graph. The signature is shown in Table 2.

We have *actualized* the graph interface by the sort `dist`, which is a parameter itself. Actualizing can be thought of as textually replacing all occurences of the identifier `label` by the identifier `dist`; for a full description see e.g. [3]. Note that parameterization in contrast to polymorphism allows not only the abstraction from sorts, but also from operations.

A prototypical implementation of `MinDist` is given by the implementation part in Table 3.

Table 2. Signature for Calculating Minimal Distances

```
SIGNATURE MinDist[dist,+,min,infinity]
  SORT dist             -- the abstract sort of distances
  FUN  +        : dist ** dist -> dist    -- accumulation
  FUN  min      : dist ** dist -> dist    -- choose
  FUN  infinity : dist                    -- worst case

  IMPORT Graph[dist]  ONLY graph node

  FUN distance : graph -> node ** node -> dist
  -- calculate the minimal distance between two nodes
```

Table 3. Implementation Part of `MinDist`

```
IMPLEMENTATION MinDist
  IMPORT Seq              COMPLETELY
         SeqMapReduce     COMPLETELY
         Graph[dist]      COMPLETELY

  DEF distance(G)(x,y) == distance(G,nodes(G))(x,y)

  FUN distance : graph ** seq[node] -> node ** node -> dist
  DEF distance(G,cand :: Rest)(x,y) ==
      -- choose between indirect path via cand and other paths
      min(distance(G,Rest)(x,cand) + distance(G,Rest)(cand,y),
          distance(G,Rest)(x,y) )
  DEF  distance(G,<>)(x,y) ==
      -- construct the sequence of weights for outgoing edges,
      -- and choose the best one
       (min,infinity) / (weight * outs(G,x))
        WHERE weight == \\edge.IF destin(edge) = y
                               THEN label(edge)
                               ELSE infinity  FI
```

A few remarks on this implementation:

- Parameterized structures can be imported omitting the actualization. Actualizations are then inferred from the context of applications of objects of the structure, a process quite similar to the instantiation of principal typings in polymorphic languages.
- In the second equation of `distance` we have made some use of the benefits of "programming with functions". The symbol \\ is the ASCII representation of the λ symbol. The λ-construct introduces a local function which returns for a given edge its weight, if it is connected to the destination. We apply

two functions from the parameterized structure `SeqMapReduce`: firstly, the map function `*`, which applies a function to each element of a sequence, is used to produce a sequence of weights; secondly, the reduce function `/`, which collapses a sequence to a single value by "accumulating" the sequence elements, chooses the best weight.

2.3 Virtual Constructors

Opal provides the TYPE item to declare an algebraic data type which is implemented later on by the DATA item. But the declaration of an algebraic type need not be identical to its implementation. Thus, it is possible to hide constructors or components of constructors, which are used for implementation-dependent formulations of exported functions. One may even use completely different sets of constructors in the interface and in the implementation. An application of this feature is given by the example in Table 4, which is taken from the Opal standard library.

Table 4. Natural Numbers in Opal

```
SIGNATURE Nat
  TYPE nat == 0   succ(pred: nat)
  -- and many more declarations

IMPLEMENTATION Nat
  IMPORT BUILTIN    COMPLETELY
  DATA nat          == abs(rep: NUM)
  DEF 0             == abs(zeroNum)
  DEF 0?(abs(x))    == eqZeroNum(x)
  DEF succ(abs(x))  == abs(succNum(x))
  DEF succ?(abs(x)) == ~(eqZeroNum(x))
  DEF pred(abs(x))  == IF succ?(abs(x)) THEN abs(predNum(x)) FI
  -- and many more definitions
```

This implementation uses the built-in integral numbers to implement natural numbers. In the interface, the TYPE item ensures the abstract properties of an algebraic free type and enables the use of pattern matching. Internally, natural numbers are as efficient as integral numbers, since simple sort embeddings as given by the DATA item are eliminated by the optimizer.

2.4 What Is Special About Opal?

We consider it essential when writing easily readable programs to have access to powerful and flexible syntactic concepts that are realized in an orthogonal manner. For this reason Opal has no built-in "syntactic sugar" for coping with special situations, but a generalization of these features which is accessible to the user:

– Identifiers in Opal consist of sequences of either alphanumeric or graphical characters. Thus, there is nothing special about the identifiers `0`, `++`, `::`, or `<>` used in the examples above.
– Function names may be placed arbitrarily before, between, or after their arguments in applications.
– Collections of objects are only separated by commas, when the order is important. Hence, parameters in function calls are seperated by commas, but declarations are not.
– Opal supports overloading in a very general way by means of Opal's *name concept*. Every object is identified by a *name* which has three components besides its *identifier*: the *origin identifier* is the name of the structure in which the object is declared, the *instance* is the list of the actualizations of that structure, and the *kind* distinguishes sorts and operations, reflecting the operation's functionality based on the names of the used sorts. The append function, for example, as applied in the implementation part of the structure `MinDist`, has the name `::'Seq[node]:node**seq'Seq[node]->seq'Seq[node]` [1].
Of course, all components of a name except its identifier can be omitted, if the context allows the unique derivation of the missing parts. (In order to perform this resolution, all available context information is used, including mutual dependencies between names.) *Note:* On the level of *names* there is actually *no* overloading; but the programmer normally uses only the *identifiers* and then has the effect of overloading.

Another important point in the design of Opal is the incorporation of software engineering aspects:

– Structures have a distinct export interface, called signature part (see Section 2.2), from which other structures can selectively import objects. To enable pattern matching over exported data types without violating the principle of information hiding, Opal incorporates the concept of virtual constructors (see Section 2.3).
– Parameterization provides a high degree of abstraction and reusability of structures (see Section 2.2). It likewise allows the abstraction from sorts as well as from operations and gives a stronger typing. The inference of actualizations of parameterized structures fits nicely into Opal's name scheme. Omitting the actualization of an import of a parameterized structure is explained as *importing the (infinite) set of possible actualizations*. The application of a partial name from such an import must contain the actualization explicitly, or it must be deducible from the context as described above.

The following feature of Opal is motivated by the desire to compile to efficient code, but it also affects the programming of functions in Opal:

– Opal is a *strict* language. Hence, infinite lists and streams cannot be expressed adequately. However, this disadvantage is not that important in the

[1] `node` is used here as short-cut for `node'Graph[dist]:SORT`, and so forth.

everyday use of functional languages. The advantage of being able to produce time- and space-efficient code is considered more valuable.

2.5 Semantics

The algebraic flavour of Opal is apparent in the semantic definition of the functional language. We presume that the reader is familiar with the basic concepts of algebraic specification languages (see e.g. [3]), and only sketch the essentials of the semantics here.

The signature and the implementation part are both assigned a specification $\mathcal{S}_e^F$ and $\mathcal{S}_i^F$ respectively. The respective signatures Σ_e^F and Σ_i^F of these specifications are related by $\Sigma_e^F \subseteq \Sigma_i^F$; the validity of this relation is ensured by Opal.

The *internal functional semantics* IntSem^F of an Opal structure $\mathcal{S}$ is the class of all $\mathcal{S}_i^F$-algebras. These algebras are defined on the basis of cpo's in the usual way. (Note: We employ a loose semantics here due to some constrained nondeterminism in guarded expressions and pattern-based definitions.)

The *external functional semantics* ExtSem^F of $\mathcal{S}$ is a class of $\mathcal{S}_e^F$-algebras that is derived from the internal functional semantics by a forget-restrict scheme: Let $A \in \mathit{IntSem}^F(\mathcal{S})$ be an internal model. Then we obtain the corresponding external model as follows: first, we form the Σ_e^F-reduct $A' = \mathit{reduct}_{\Sigma_e^F}(A)$, then we extract the subalgebra $A'' = \mathit{restrict}_{\Sigma_e^F}(A')$ of reachable elements w.r.t. Σ_e^F-operations. A'' then belongs to $\mathit{ExtSem}^F(\mathcal{S})$.

This distinction of the external semantics by a forget-restrict process is a feature we have borrowed from the area of algebraic specification. It has turned out to be mandatory for any reasonable development and verification methodology. Whether an additional *identify* operator should be used for the external functional semantics is a question we consider worthy of further research.

3 The Property Language

As mentioned earlier, we aim at an integration of specification concepts and functional concepts. "Pure" specification languages focus on expressing algebraic properties of one or several stages of the software development process. Hence, the need for executable specifications is only a minor point in the concepts of these languages and efficiency is not taken into consideration at all.

In contrast, the concept of Opal is just the other way round: we concentrate on the production of executable, efficient software, but are therefore more restrictive in our specification features. To emphasize this difference in motivation, we use the term Opal "property language" rather than Opal "specification language".

3.1 Laws

Properties are expressed by first-order predicate-logic formulas. The primitive predicates are congruence (===) and definedness (DFD) of functional expressions.

Assume we have a function `connected` for graphs. We can specify the behavior of this function as follows:

```
FUN connected: graph -> node ** node -> bool
LAW ALL G x y.connected(G)(x,y) <==>
      Y in (destin * outs(G,x)) OR
      (EX z. z in (destin * outs(G,x)) AND connected(G)(z,y))
```

We would like to point out the following:

- We distinguish boolean values in the functional language from truth values in the logical language. This is necessary because the former express *computable* values, whereas the truth values are not necessarily computable. Nevertheless, we have introduced an abbreviation feature for the denotation of boolean expressions in places where formulas are expected. Actually, the above formula reads: ...`connected(G)(x,y) === true <==>` ...

- Note the difference between the congruence symbol `===` and the definition symbol `==`. The former denotes strong equality (which yields true for two undefined values, and is in general not computable), whereas the latter denotes *fixpoint* equality. Hence, the definition DEF `f(x) == f(x)` sets `f` to the least fixpoint which satisfies the equation (i.e. is the function which is totally undefined), while LAW ALL `x. f(x) === f(x)` is a tautology and therefore does not say anything about `f` at all.

3.2 Structures

Following the same software engineering principles as for the functional sublanguage, we again distinguish two parts of properties:

$$\text{properties} \quad = \quad \text{external properties} \quad + \quad \text{internal properties}\ .$$

As is to be expected, the external properties are the only ones that are available to the environment, whereas the internal properties express facts about implementation details and therefore must be hidden from the environment.

For example, in the structure `MinDist` we might have an external property part which simply expresses the fact that the function `distance` is total:

```
EXTERNAL PROPERTIES MinDist
  LAW ALL G x y. DFD distance(G)(x,y)
```

Of course, it is possible to also express the input-output behavior of the function `distance`. But in general it is probably quite sufficient in our framework to formally specify only those properties of functions which interest us because of special safety requirements the produced software has to obey.

The internal property part of a structure may be used to express certain facts about the implementation of a structure. This can be used to express facts about auxiliary functions not visible in the external view of a structure, or to hide some facts which are not essential for the external view. For example, we might express the internal property:

```
INTERNAL PROPERTIES MinDist
  LAW ALL G x y. NOT connected(G)(x,y) ==>
                                      distance(G)(x,y) === infinity
```

This property may be used for example by an optimizer to enhance applications of the function `distance` in contexts where preconditions ensure that `x` and `y` are not connected.

Note that due to the semantics given in Section 2.5, the implementation does not necessarily satisfy the properties given in the external property part. That is, some properties given in the interface rely on the fact that the forget-restrict operation has been performed.

For example, the sort `graph` might be implemented by adjacency matrices, using a DATA declaration like DATA `graph == graph(adj:matrix[bool])`. While the functions in the interface guarantee that the adjacency matrix is always quadratic, the internal constructor `adj` does not enforce this. So the definedness property of `distance` stated above is not valid in the implementation part, since the application of the function `distance` to a graph with a nonquadratic adjacency matrix is not defined.

One might wonder why property parts are conceptually distinguished parts of a structure, and are not merged with the signature or implementation part. This is motivated as follows:

- The external as well as the internal property parts require auxiliary functions — just as the implementation often requires auxiliary functions. But for specifying properties, other auxiliary functions may be needed than in the implementation part. There might even be a necessity to specify functions which are not intended to be implemented. Therefore, it makes sense to separate these functions from the implemented auxiliary functions in the implementation part.
- We would like to enable the user to view the functional part of a structure without its property parts.

3.3 Semantics

Now we have a situation where there is a specification as well as a functional program, and this necessitates certain compatibility requirements. Therefore we will first sketch the semantics of the property parts (which is a standard loose semantics) and then consider the compatibility criteria.

The relationship between the declarations of the several parts of an Opal structure is documented by the following diagram:

Signature $\mathcal{S}^F_e$	*External Properties* $\mathcal{S}^P_e$	visible
Implementation $\mathcal{S}^F_i$	*Internal Properties* $\mathcal{S}^P_i$	hidden

Opal automatically includes the signatures from the visible into the corresponding hidden parts and from the functional parts into the corresponding property parts. Thus, the signatures are related by $\Sigma^F_e \subseteq \Sigma^F_i$, $\Sigma^F_e \subseteq \Sigma^P_e$ and $\Sigma^F_i \subseteq \Sigma^P_i$, $\Sigma^P_e \subseteq \Sigma^P_i$.

The *external property semantics* $ExtSem^P$ of an Opal structure $\mathcal{S}$ is constructed from the (loose) class of $\mathcal{S}^P_e$-algebras. On this class of algebras we perform the reduct operation with respect to the signature of the signature part.

The *internal property semantics* $IntSem^P$ is analogously defined as the Σ^F_i-reduct of the class of $\mathcal{S}^P_i$-algebras.

The notion of *model correctness* states how the property semantics of an Opal structure is related to the functional semantics:

- *Internal model correctness:* $\emptyset \neq IntSem^F(\mathcal{S}) \subseteq IntSem^P(\mathcal{S})$
- *External model correctness:* $\emptyset \neq ExtSem^F(\mathcal{S}) \subseteq ExtSem^P(\mathcal{S})$

Note that for model correctness, the connection between the internal and external property part is only established implicitly through their common relationship to the implementation. However, correctness can only be determined if a complete implementation of the structure is present.

In order to enable the evolutionary development of implementations by correctness-preserving transformations, we envisage an additional notion of correctness which allows for an implementation being only partially present. The situation here is complicated by the constrained nondeterminism of the functional language Opal: we cannot simply amalgamate properties with definitions from the implementation, nor can we just construct the intersection using the loose class of algebras in $IntSem^F(\mathcal{S})$ and the loose class of algebras in $IntSem^P(\mathcal{S})$, since the first one stands for "don't know which model is chosen", whereas the second stands for "don't care which model is chosen". This subject is currently a topic of ongoing research.

4 Compiling Algebraic Programming Languages

For algebraic programming to become feasible in practice, it is of fundamental importance that the resulting programs are executed efficiently enough to be competitive with programs written in more machine-oriented languages such as C or Pascal. The *basic* compilation of functional languages has been a topic of intensive research during the past decade. The results are encouraging, although not totally satisfactory. The *extended* compilation of algebraic languages is a topic of ongoing research in which we invest considerable effort. The idea is, basically, to exploit algebraic properties for advanced compilation schemes.

4.1 Basic Compilation

Under the catchword "basic compilation" we understand the mapping of functional programs to von-Neumann-like machine architectures *without* ambitious structural transformations of the functional source. The problems in the basic compilation are the treatment of recursion, functional values, lazy evaluation, and recursive data types, particularly with respect to storage reclamation ([6, 11, 12]).

For OPAL we take the approach of compiling to the "high-level assembly language" C instead of some concrete machine architecture. This has several advantages: the resulting code is highly portable, and we can benefit from many of the optimization techniques for imperative languages nowadays performed by C compilers. Moreover, we have a well-defined interface for interlanguage working, which allows us to access the huge amount of existing standard software available in C.

On the way from OPAL to C code several analysis and transformation phases are invoked. We merely sketch some of them:

- The *Import Analysis* collects information for system global optimizations, such as definitions and properties of the implementations of imported structures. This phase is usually only activated for ready-to-ship systems, since it creates recompilation dependencies between structure implementations.
- The *Unfold Analysis* examines which functions have to be unfolded and in which order. Unfolding is a prerequisite for common subexpression elimination and simplification.
- The *Common Subexpression Elimination* phase detects all common subexpressions local to functions. Note that the potential for common subexpressions is very large in functional languages, because of the absence of side-effects.
- The *Simplification* phase performs fusion and tupling of function local computations.
- The *Translation* phase translates to intermediate imperative code, mainly establishing the notion of *memory* and reference. This is achieved by using a reference-counting scheme.
- The *Selective Update / Reusage* analysis introduces the immediate reclamation of released memory (instead of returning it to the free memory pool), and in particular cases, the *selective updating* of data objects. In our framework, a selective update is performed if a released memory cell is immediately reused for constructing a new cell, and some of its components should be simply copied to the new cell – then we can omit copying (see example below). The reuse analysis is based on static and dynamic reference-counting information.
- The *C Code Generation* phase finally produces the C target code. This is also the place where several forms of recursion (tail recursion, tail recursion modulo constructor application) are mapped to iteration (see example below).

We include a short example of the generated C code, which mainly illustrates the effect of the selective update / reusage optimization, the compilation of recursion, and the handling of functional values. The example is the filter function on sequences as shown in Table 5. The compiler generates the (hand-

Table 5. The Filter Function on Sequences

```
FUN filter : (data -> bool) -> seq[data] -> seq[data]
DEF filter(P)(ft::Rt) == IF P(ft) THEN ft :: filter(P)(Rt)
                                  ELSE      filter(P)(Rt) FI
DEF filter(P)(<>)     == <>
```

formatted and commented) C code given in Table 6. We would like to point out the following properties of the generated C code:

- `RELEASE`, `FIELD` etc. are of course C macros which expand to C statements.
- The function `filter`, although not tail-recursive in nature, is compiled to an iterative loop. This is achieved by introducing the "result pointer" `resp`, which always points to the place where the result of the next iteration shall be stored.
- In case that the processed sequence is exclusive – i.e. each cell in the linked list representation has a reference count of one – no new list elements are allocated, and only the link to the next cell is occasionally updated. Hence, the only overhead in this case compared to an imperative program is the test of the reference counter, which is compiled to a single machine instruction.

Just to give a brief impression of the attainable efficiency, we mention two classical benchmarks, viz. the functions *Tak* and *Tree* (originally suggested in [14]). *Tak* tests the behaviour of recursion, while *Tree* is a typical program which manipulates recursive data structures. We compare the Opal compiler with Standard ML of New Jersey, Version 0.65, and also include benchmarks of versions of the algorithms written in (moderately hand-optimized) C [2], compiled with the GNU C-Compiler, Version 1.4, on a SUN-4/75 (32MB):

	C	Sml	Opal
Tak	3.9	35.910	4.415
Tree	2.045	6.110	1.228

For further details about our storage optimization and garbage collection techniques, we refer to [12] and [11].

[2] The C version of the *Tree* benchmark uses the memory allocation routines from the standard library. It make uses of imperative destructive update.

Table 6. Generated C Code for the Filter Function on Sequences

```
OBJ filter(OBJ P,OBJ S)
{OBJ res; OBJ *resp=&res;        /* resp points to the location where to
                                    store the result of the next iteration. */
 for(;;){
  if(IS_PRIMITIVE(S)){           /* S is a primitive value -- in our context
                                    this must be the empty list. */
   RELEASE(P,1);                 /* release 1 reference to closure P. We do not
                                    need to release S since it is primitive. */
   *resp=_Slg;                   /* '_Slg' is the alphanumerical */
   break;                        /* representation of '<>' */
  }else{
   {OBJ Ft = FIELD(S,1);         /* select ft and rt components */
    OBJ Rt = FIELD(S,2);
    int Excl_S = IS_EXCL(S));    /* Check if we are the only reference to the */
    OBJ T1,T2;                   /* cell S. */
    if (Excl_S){
      RESERVE(Ft,1);             /* We plan to consume 2 references to Ft and */
    } else {                     /* 1 to Rt (see below). But in the exclusive */
      RESERVE(Ft,2);             /* case we can 'borrow' some from S. */
      RESERVE(Rt,1);
    }
    RESERVE(P,1);
    T1 = METHOD(P,1)(P,Ft);      /* Each closure carries an array of methods
                                    describing how to evaluate it with N
                                    parameters. Here, N=1.
                                    Now we have consumed Ft the 1st time! */
    if (IS_TAGGED(T1,1)){        /* Test if the predicate yields true */
      if (Excl_S){
        *resp = S;               /* We can reuse S. And moreover, Ft is  */
      } else {                   /* already at its place. */
        RELEASE(S,1);
        *resp = CREATE(2);       /* Create cell of size 2, and initialize it. */
        FIELD(*resp,1) = Ft;     /* This is where Ft is consumed the 2nd time.*/
      }
      S = Rt;                    /* Prepare next iterate, consuming Rt. */
      resp = &FIELD(*resp,2);    /* Setup location to store result of  */
      continue;                  /* next iteration. */
    } else
      if (Excl_S){
        DISPOSE(S,1);            /* Dispose the cell S refers to. Note that */
                                 /* the components are already released */
     } else {                    /* indirectly (we have consumed them). */
        RELEASE(S,1);            /* Release the reference S. */
      }
      S = Rt;                    /* resp still points to the place where
                                    the result of the next iteration
      continue;                     shall be stored. */
    }
  }
  return res;                    /* return front of the filtered list. */
}
```

4.2 Extended Compilation

Even though the basic compilation performs already quite nicely, our ambitions go further. We want to apply more elaborated optimization tactics on the source level, such as extended recursion removal, function combination, function composition, partial evaluation, finite differencing, and so forth ([8]). None of the existing compilers for functional languages applies these techniques. The main reason is that these tactics usually require information about the program which is not deducible from the program text alone — at least not automatically by means of a compiler. Hence, the additional information has to be incorporated into the compilation by the programmer as an "advice" to the compiler. This approach has been called *extended compilation* ([4]).

In our setting additional information can be naturally expressed by algebraic properties. Moreover, optimization tactics themselves can be expressed by *algorithm theories* [13], leading to a knowledge-based extendable compilation system. We illustrate the principal ideas of this approach by an example.

The following function computes the chromatic number of a graph G.

```
DEF chromaticNumber(G) ==
   IF complete?(G) THEN cardinality(nodes(G))
                   ELSE min(chromaticNumber(amalgamate(G)),
                            chromaticNumber(connect(G)))    FI
```

Here, the functions amalgamate and connect choose the next pair of unconnected vertices in the graph, and identify or connect them, respectively. Clearly, one of the efficiency problems of this algorithm is that each recursion level creates modified versions of the graph– this is in particular painful if the graph is represented by a monolithic data structure such as an adjacency matrix. Since the graph is shared between the two recursive incarnations, the compilation techniques for selective updating of data structures cannot be applied. A better solution would be based on a backtracking algorithm, which undoes the modifications at each recursion level and uses only one "single-threaded" instance of the graph.

The compiler would be capable of generating this algorithm automatically, if it is provided with a suitable theory of backtracking. One possible backtracking theory matching our problem is given in Table 7. It is expressed as an ordinary parameterized OPAL structure with properties. *Applying* the theory BackTrack to our problem means finding a semantically correct instantiation of the parameter, such that function f is instantiated with function chromaticNumber. Once having found this instantiation, it is a simple matter of term rewriting to replace applications of chromaticNumber by the function bt (using the LAW bt_law), and to apply specialization techniques to simplify the implementation of the definition of bt according to the concrete instantiation.

The least information the compiler requires for finding an instantiation automatically is the following:

```
   FUN unconnect    : graph -> graph -> graph
   FUN unamalgamate : graph -> graph -> graph
   LAW ALL G . (unconnect(G)    o connect   )(G) === G
   LAW ALL G . (unamalgamate(G) o amalgamate)(G) === G
```

Here, the function unamalgamate extracts the necessary information from a graph to construct a function, which undoes an amalgamation on the given graph; the function unconnect behaves similar. Of course, these functions must be implemented by the programmer such that they behave as specified.

Table 7. A Backtracking Theory

```
SIGNATURE BackTrack [f,A,B,b,t,h,d1,d2,inv1,inv2]
  SORT A B                              -- parameter
  FUN f t   : A -> B    b : A -> bool   h : B ** B -> B
      d1 d2 : A -> A    inv1 inv2 : A -> A -> A
  FUN bt : (A -> A) ** A -> A ** B  -- introduced function

EXTERNAL PROPERTIES BackTrack
  -- properties of the parameter
  DEF f(x) == IF b(x) THEN t(x) ELSE h(f(d1(x)),f(d2(x))) FI
  LAW inv1 == ALL x . (inv1(x) o d1)(x) === x
  LAW inv2 == ALL x . (inv2(x) o d2)(x) === x
  -- properties of introduced function
  LAW bt_law == ALL x . f(x) === y WHERE (_,y) == bt(\\z.z,x)

IMPLEMENTATION BackTrack
  DEF bt(inv,x) ==
      IF b(x) THEN (inv(x),t(x))
      ELSE bt(inv1(x),d1(x))   ; (\\x1,y1.
           bt(inv2(x1),d2(x1)) ; (\\x2,y2.
           (inv(x2),h(y1,y2))              ))
             WHERE ; == \\a,b,C. C(a,b)         FI
```

A sketch of a possible implementation looks as follows:

```
DEF unconnect(G) ==
    LET (u,v) == unconnectedVertexPair(G)
    IN \\NewG.removeEdge(NewG,u,v)
DEF unamalgamate(G) ==
    LET (u,v) == unconnectedVertexPair(G)
        ModifiedEdges == <<  all edges touching u or v >>
    IN \\NewG.restoreEdges(NewG,ModifiedEdges)
```

Note the use of higher-order functions to "store" the information on how to undo a modification of the graph[3]. Clearly, the function `unamalgamate` as given here is in the worst case as expensive as copying a graph; depending on the underlying implementation of graphs, more efficient versions are possible.

Given these functions and properties, the compiler can now syntactically prove that the following is a correct instantiation of the backtracking theory:

[3] Technically, when calling e.g. `unamalgamate(G)` this information is stored in a closure as an implicite parameter to the anonymous function returned.

```
IMPORT BackTrack [chromaticNumber,graph,nat,complete?,
                  \\G.cardinality(nodes(G)),min,
                  amalgamate,connect,unamalgamate,unconnect]
```

The proof is performed by syntactic unification techniques. More generally, for each function definition the compiler tries to instantiate from a set of given algorithm theories. This operation is still very expensive, and there are several approaches under investigation to guide and improve this search. A promising one is the use of *property theories*. Applied to our example, the properties of an inversion function would be put into a distinguished structure:

```
THEORY LocalInversion[A,f,i]
  SORT A   FUN f : A -> A   i: A -> A -> A
  LAW ALL x. (i(x) o f)(x) === x
```

Now, instead of the axioms `inv1` and `inv2` this theory would be instantiated in `BackTrack`, and instead of giving the properties of `unconnect` and `unamalgamate` explicitly, the programmer would instantiate this theory as well[4]. A syntactic proof that the inversion properties hold now simplifies to a check whether a particular theory has been instantiated. The goal of this approach is to find a small set of property theories such as `LocalInversion`, `Monoid`, `Lattice` etc. which is shared by a large amount of algorithm theories.

There are several challenges for future research in the area of extended compilation, as it is applied to algebraic programming. On a conceptual level, algorithm and property theories suitable for this approach have to be recovered, collected and systematized. On an engineering level, the implementation technology has to be worked out by combining methods from term rewriting, higher-order unification, and classical compiler construction.

5 Conclusion

We feel that there is great leverage to be gained from making maximum use of the amalgamation of functional and algebraic paradigms. It is by now a well-known thesis that functional programs can be formally derived from algebraic specifications. The research project KorSo sets out to convert this thesis into practically applicable methodology, including tool support. Whereas other languages used in this project mainly focus on the aspect of developing specifications, Opal fosters the transition from specification to executable (functional) code.

Currently, Opal is used in teaching Compiler Construction and Software Engineering Principles at our university[5]. Furthermore, it is employed in a joint

[4] The integration of these kinds of theories into Opal, which are used only for bracketing algebraic properties, but do not represent an executable piece of the program, is currently under investigation.

[5] The Opal compiler is available by anonymous ftp from `ftp.cs.tu-berlin.de`, directory `pub/local/uebb/ocs`. For comments and requests please contact us at the e-mail address `opal@cs.tu-berlin.de`.

project with Daimler-Benz AG in implementing and verifying a compiler for a programmable controller language.

But there is more to be gained from the fusion of algebra and functional programming. An elaborate usage of algebraic properties can generate optimizations that possibly go far beyond the capabilities of classical compilation. And this may help to take functional and algebraic approaches from academic prototype-building to the practical software production stage.

Acknowledgement

We thank our colleagues from the Opal project and from the KorSo project for many stimulating discussions. This also applies to our former colleagues Gottfried Egger, Michael Jatzeck and Wolfram Schulte. Niamh Warde and Manuela Weitkamp-Smith provided valuable assistance in formulating the paper.

References

1. F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gantz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Semelson, M. Wirsing, and H. Wössner. *The Munich Project Cip*, volume 1. LNCF Springer, Berlin, 1985.
2. I. Classen. Semantik der revidierten Version der algebraischen Spezifikationssprache ACT ONE. Technical Report 88/24, TU Berlin, 1988.
3. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications I, Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science 6, Springer, Berlin, 1985.
4. M.S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Program Specification and Transformation.* North-Holland, 1987.
5. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. POPL*, 1985.
6. Simon L. Peyton Jones. *Implementing Functional Languages.* Prentice Hall, 1992.
7. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
8. H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer–Verlag, Berlin, 1990.
9. P. Pepper. The Programming Language OPAL. Technical Report 91–10, TU Berlin, June 1991.
10. N. Perry. Hope+. Internal report IC/FPR/LANG/2.51/7, Dept. of Computing Imperial College London, 1988.
11. W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Declarative Programming.* Springer Verlag, 1992.
12. Wolfram Schulte. *Effiziente und korrekte Übersetzung strikter applikativer Programmiersprachen.* PhD thesis, Technische Universität Berlin, 1992.
13. Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305–321, 1990.
14. R. Stansifer. Imperative versus functional. *SIGPLAN Notices*, 25, 1990.

This article was processed using the LaTeX macro package with LLNCS style