

**Title** Composition by explicit continuations

**Abstract**

Direct composition of functions is well-known, but has some shortcomings as a model of software component composition. A better model is obtained with *continuations* which were initially used to model some of the nastier programming concepts (like goto). This paper provides justifications for this position, illustrates it by examples, and tells briefly about a tool that supports it.

**Author** Jørgen Steensgaard-Madsen

**Affiliation** Technical University of Denmark

**Postal address**

Jørgen Steensgaard-Madsen  
DTU/IMM Building 322  
Richard Petersens Plads  
DK-2800 Lyngby  
Denmark

**email** [jsm@imm.dtu.dk](mailto:jsm@imm.dtu.dk)

**Phone number** (+45) 4525 3732

# Composition by explicit continuations

Jørgen Steensgaard-Madsen, DTU

**Abstract.** Direct composition of functions is well-known, but has some shortcomings as a model of software component composition. A better model is obtained with *continuations* which were initially used to model some of the nastier programming concepts (like goto). This paper provides justifications for this position, illustrates it by examples, and tells briefly about a tool that supports it.

Given functions  $f, g : \text{Datastream} \rightarrow \text{Datastream}$  and a value  $\text{input} : \text{Datastream}$  direct composition is expressed by  $f(g(\text{input}))$ . The similar composition with (explicit) continuations could be expressed as follows

```
program{
  g'(input) x;
  f'(x.value)y;
  output << y.value;
}
```

where  $f'$  and  $g'$  are operations similar to the functions  $f$  and  $g$ . The formulation agrees with an object-oriented view of programmers. For a theoretically minded it might be understood as

$$g'(\text{input}, C_g)$$

where  $C_g = \mathbf{fn}(\text{value}) \Rightarrow f'(x.\text{value}, C_f)$   
where  $C_f = \mathbf{fn}(y.\text{value}) \Rightarrow \text{output} \ll y.\text{value}$

Obviously, this depends on composition of functions, but it uses a different pattern, and  $g'$  requires an extra argument, a continuation  $C_g$ , compared to  $g$ . Another way of saying it is that  $g'$  has its own composition operation, represented by the presence of parameter  $C_g$ . Yet another is to say that *results are given to a receiver*, rather than *returned*.

The problem with functional composition arises when one considers pairs like  $(T:\mathbf{type}, x:T)$  as values. We expect that such values can be decomposed, and afterwards it is hard to maintain the close ties between the components. This is much easier, when the pair is passed to an argument function of type  $[T]T \rightarrow W$ , or more generally  $[T]\Phi_T \rightarrow W$  with  $\Phi_T$  representing an interface that depend on some unknown type  $T$ . Such an argument is continuation-like for which well-known scope rules of second-order, typed lambda calculus apply and the problems with the problematic pairs essentially vanish.

The notions of classes and objects are heavily burdened with attributes that are hard to underpin theoretically. We have found it attractive to use simplified notions that are theoretically supported. As illustrated, some of the notational convenience of object-oriented programming can be maintained. The resulting focus will be on class usage rather than on definition.

A class-like component,  $C$ , may have (function-)type  $([T]\Phi_T \rightarrow W) \rightarrow W$ . We identify the scope of the members of  $C$  with the continuation of an object instantiation and make it explicit in a statement like

```
C myobj { ... myobj.member_k(...); ... };
```

Furthermore, we allow use of an ‘instantiation notation’ as an alternative, semantically equivalent:

```
program{ C myobj; ... myobj.member_k(...); ... };
```

This view restricts the usual notion of objects a little (objects are not values) and allows for a generalisation to operations with several continuation-like arguments, i.e. operations with function type like

$$([U]\Phi_U \rightarrow W) \rightarrow ([V]\Phi_V \rightarrow W) \rightarrow W$$

This ties the notion of object instantiation to program statements and class definitions to semantics of statements.

We maintain that a modularised system must depend on some language of composition, although it may not be clearly distinguished from the language used for module definition. Furthermore, that composition languages can be made explicit and considered as an interesting family of domain specific languages.

Thus we find that

1. Components may provide semantics for a general notion of statements
2. Composition may adhere to familiar rules for

*Expressions, Structured statements, and Blocks*

The scope of member names of an object forms an *explicit continuation* of an object

*implicit continuations* ( $\sim$  'rest of the program') are useful in formal semantics

*explicit continuation* (clauses  $\sim$  'branches' or 'bodies') are useful for components

We have developed a tool that technically combines well-known concepts:

some continuations are explicit clauses of statements

an object instantiation and its explicit continuation is a statement with one clause

names of any kind may be implicitly bound (like members) in a clause

a statement may have several subordinate statement clauses

The tool, *dulce*, is intended for implementation of domain specific languages, as are programs like *lex* and *yacc* and many others. Such tools aim to solve the core problem of language implementation, as given by

$$(G \times S) \rightarrow P \rightarrow I \rightarrow O$$

where  $G, S, P, I$  and  $O$  denote *grammar*, *semantics*, *programs*, *input* and *output*, respectively. We split the  $G \times S$  part into a *deep*- and a *surface* structure

$$\begin{aligned} (G_D \times S_D) &\rightarrow (G_T \times S_T) \rightarrow P \rightarrow I \rightarrow O \\ (g_d, s_d) &\mapsto \textit{dulce}: (G_T \times S_T) \rightarrow P \rightarrow I \rightarrow O \end{aligned}$$

We can do this so that the following properties hold

$$\begin{aligned} \textit{dulce}(g_{1,T}, s_{1,T}) &\subset \textit{dulce}(g_{1,T} \cup g_{2,T}, s_{1,T} \cup s_{2,T}) \\ \textit{dulce}(g_{2,T}, s_{2,T}) &\subset \textit{dulce}(g_{1,T} \cup g_{2,T}, s_{1,T} \cup s_{2,T}) \end{aligned}$$

which means that implementations are build like sets, i.e. incrementally.

Users benefit from this split in two steps because the last does not require any expertise in parsing, type checking, and many other related notions. Lisp-like languages have a similar split: their deep structure consists of lists denoted by S-expressions, and their surface structure is given by a set of predefined operations. Our deep structure is not directly accessible to programmers, but it is denoted by a program-like common notation.

Various approaches to modularisation have been investigated over many years, for instance classes in Simula [2], modules in Modula [7], and functors in ML [1]. The approach taken here pursues an approach advocated by the author a long time ago as a statement-oriented approach [5]. It depends on some of the important notions and techniques of other approaches, in particular the notions of types, polymorphism, and type inference in ML. In addition it is closely related to an analogy between proofs and programs [3], and our notion of components is closely related to the notion of weak existence in logic [4].

# 1 Description and composition of components

A formal description of components follows the pattern that can graphically be presented as

$$\begin{array}{c}
 \text{operation\_name : type} \quad \frac{\frac{\text{resource}_1}{\text{resource}_2}}{\text{resource}_3} \\
 \dots
 \end{array}$$

but actually a text representation is used.

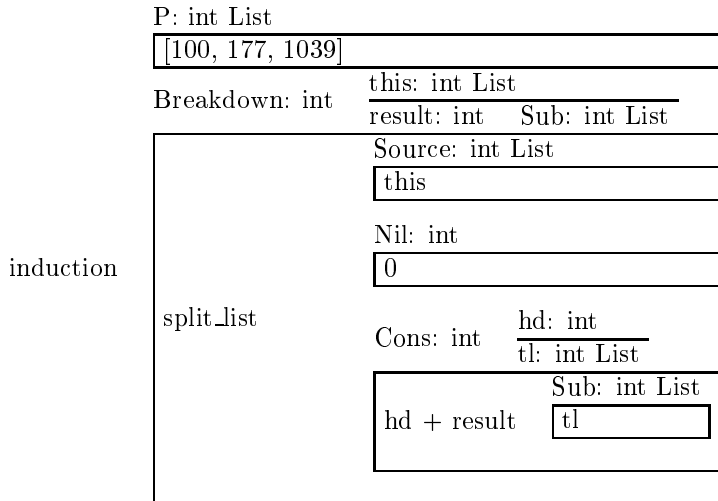
The resources are described in precisely the same way. The resources are like operands or arguments needed for the operation to be performed. Two examples:

$$\begin{array}{c}
 \text{induction:Result} \quad \frac{\text{P:Problem}}{\text{Breakdown:Result} \quad \frac{\text{this:Problem}}{\text{result:Result} \quad \text{Sub:Problem}}}
 \end{array}$$

$$\begin{array}{c}
 \text{split\_list:W} \quad \frac{\frac{\text{Source:T List}}{\text{Nil:W}}}{\text{Cons:W} \quad \frac{\text{hd:T}}{\text{tl:T List}}}
 \end{array}$$

The first is modelled over mathematical induction and the second combines a branch on the structure of a list and a decomposition of the list when it is non-empty.

The two components can be used in combination to express summation of numbers in a list as follows



Each required resource (i.e. P and Breakdown for induction) denotes a computation which must be provided in a box inside which a resource of the resource can be used (e.g. this and result inside the box matching Breakdown). In addition the types are instantiated for the task at hand, as described in the next section.

Actually we use a linear text-representation where the description of resources do not appear as illustrated below. The descriptions are absent because they are considered part of the grammar. The text representation of the combined use of induction and split\_list above is simply:

`induction{[100,177,1039]}{split_list(this,0,hd+result(tl))}`

Compositions may require sequences of applications, for which the text-representation can be outlined as follows

induction{[100,177,1039]}{ ...; split\_list(this,0,hd+result(tl)); ... }

The formal description of a component being a class is a special case

$$\begin{array}{c}
 \frac{\text{class\_name} : \text{void} \quad \text{Continuation} : \text{void} \quad \frac{\frac{\frac{\text{member}_1}{\text{member}_2}}{\text{member}_3}}{\dots}}{\dots}}{\dots}
 \end{array}$$

where the only resource identified by *Continuation* constitutes the scope of object-members. The name has been chosen to stress the fact that resource definitions are continuations that receives the (right to use) the resources of the resource it defines.

The rules for textual representation admit two pattern of applications that are equivalent

$$\text{class\_name } x \{ \dots x.\text{member}_k; \dots \} \quad \equiv \quad \begin{array}{l} \text{program}\{ \\ \text{class\_name } x; \\ \dots x.\text{member}_k; \dots \\ \} \end{array}$$

The characteristic of our approach to components and their composition is their dual generalisation of both classes and statements.

## 2 Types

Types acquire the rôle of grammatical categories, i.e. a typical category like *Expression* is split up into  $\mathcal{T}$  *Expression* with  $\mathcal{T}$  varying over types. This allows operator symbols to be overloaded and meanings to be determined by the observed types of operands.

Components are typed, and types play an important rôle in their descriptions. A description may be generalised on types so that it can be instantiated for types for the case at hand. The generalised descriptions follow a pattern that can be given graphically as

$$\begin{array}{c}
 \text{formal\_type\_operator}_i \\
 \text{formal\_type\_operator}_{ii} \\
 \dots
 \end{array}
 \left| \begin{array}{l} \\ \\ \\ \end{array} \right.
 \begin{array}{c}
 \frac{\text{resource}_1}{\text{resource}_2} \\
 \frac{\text{resource}_2}{\text{resource}_3} \\
 \dots
 \end{array}
 \begin{array}{l}
 \text{operation\_name} : \text{type} \\
 \\ \\
 \end{array}$$

Type operators may require type operands, as for instance List.

Both examples depend on generalised descriptions, e.g.

$$\begin{array}{c}
 \text{T} \\
 \text{W}
 \end{array}
 \left| \begin{array}{l} \\ \\ \\ \end{array} \right.
 \begin{array}{c}
 \text{Source:T List} \\
 \hline
 \text{Nil:W} \\
 \hline
 \text{Cons:W} \quad \frac{\text{hd:T}}{\text{tl:T List}} \\
 \hline
 \hline
 \end{array}$$

which is instantiated for the composition by substitution of int for both T and W. This instantiation is done by type inference<sup>1</sup> in *dulce*, i.e. users do not have to worry about types, except when error messages appear.

<sup>1</sup>*dulce*'s type system depends on universal quantification over type operators, not necessarily of arity 0. The detailed properties of our inference algorithm have not been fully clarified. *dulce* regrets in situations where higher-order inference is required.

A type operator may be introduced as a resource, e.g. with a continuation as its scope, just like members of an object. A fundamental symmetry allows resources to be described exactly as operations. It essentially provides the means for a component to introduce new type operators, e.g. List or Set. They are ‘member type operators’ similar to ‘member functions’.

A type operator that is part of a resource description has to be instantiated also, when the operation that requires the resource to be defined is applied. In that case type operator names are replaced by unique names distinct from others. This is necessary to distinguish between several instances of the type operator, arising from each application of the operation.

### 3 Applications

The *dulce* system has been used to build several language contributions or components above the level of ordinary operators and operations/statements. Quite a few for interpreters that produce text which has to be used by other interpreters, i.e. they have the flavour of program generators:

One component is for simple production of html-documents [6]. This is convenient when combined with computations that determine part of the contents of the final document.

Another is used to generate multiple choice problems to be submitted to a server for checking and feedback generation. This of course uses the one above.

One component has been written to illustrate translation of a program to the fundamental parts of a hardware realisation of it

Components are being written to generate texts as used by the xfig program to represent line drawings. Like the html-document component it should be convenient to generate drawings with a substantial element of algorithmically determined parts, e.g. graphs of functions and syntax diagrams

### 4 Tool support

The tool *dulce* has been developed as a proof of feasibility of our notions of composition and component. It can be used for various tasks, as we shall now describe.

First *dulce* is used to build interpreters for composition languages. A language can be considered characterised by a collection of components and the programs of the language as expressions of component compositions. The textual examples given earlier illustrate the kind of expressions that can be interpreted.

Second, it is used to build components from descriptions as illustrated. An outline of a component implemented in C can be generated from the description. Often the component will be a wrapper that applies facilities of another kind, e.g. library routines.

Third, the interpreters built with the tool can be used to translate composition expressions into C. This can be done with no more semantics of components as the outlines that can be generated from component descriptions.

Use of *dulce* is controlled by actions invoked through the `make` program. This assumes that a shallow hierarchy exists for development of languages. At the top of the hierarchy reside a few common definitions. At the next level contributions are built with a contribution being a collection of components. For each contribution a language for testing it will be appropriate. As a consequence a final language can be constructed as a test language for an empty contribution.

The simplicity of use can be illustrated by showing the information actually in a contribution subdirectory to build a test language.

```
ULCSYS = demo
PARTS  = operators types statements utilities \
        variables files sets multiprogramming
EXTRALL = -lpthread -lcrypt
include ../makefile.include
```

The first line provides a name for the test language, the second lists other contributions that should be included, the third informs about the libraries that are needed, and the final one includes common definitions for make. These lines contain, literally, all the information needed to automatically build a language from the eight separately developed components.

## 5 Summary

We have illustrated a number of points by actual realisations. In summary, with some additional strength added, the essential points are

### Language structure

**Some languages have both a deep and a surface structure**

**The surface structure of a language can be a collection of components**

**The deep structure of a language may handle component composition**

### Engineering implications

**System construction is a special case of language implementation**

**Application experts should define components, not languages**

**Experts should contribute to several domain specific languages**

### Components and statements

**Components generalise both statements and classes**

**The scope of an object is the continuation of its instantiation**

**Components may have multiple continuations: *views* are *branches* (?)**

## References

- [1] A. Appel. Standard ML reference manual, 1988.
- [2] O.-J. Dahl, B. Myrhaug, and K. Nygaard. The simula 67 common base language. Technical report, Norwegian Computing Center, 1968.
- [3] William A. Howard. The formulae-as-types notion of construction. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [4] J. Mitchell and G. Plotkin. Abstract types have existential type. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 37–51. ACM, Jan. 1985.
- [5] J. Steensgaard-Madsen. A statement-oriented approach to data abstraction. *ACM Transactions on Programming Languages and Systems*, 3(1):1–10, January 1981.
- [6] Jørgen Steensgaard-Madsen. HTEL: a HyperText Expression Language. *Software – Practice and Experience*, 29(8):661–675, 1999.
- [7] N. Wirth. Modula: A language for modular multiprogramming. *Software – Practice and Experience*, 7(1):3–35, January&February 1977.