The
Higher-Object Programming System
"HOPS"

Arne Bayer and Wolfram Kahl

Fakultät für Informatik
Universität der Bundeswehr München
D–85577 Neubiberg, Germany

email: {arne|kahl}@informatik.unibw-muenchen.de

Abstract

Herein, a graphical tool (HOPS) for the manipulation of directed acyclic term graphs (DAGs) is described. Programming in a truly functional style is facilitated by the provision of immediate user interaction with functional program DAGs, a novel approach to generic data type construction and a powerful and flexible transformation system. Conceptual decisions in the design of language and system and the way they influence the programmer are outlined.

1 Introduction

The Higher Object Programming System (HOPS[1]) described here has its roots in the prototypes developed initially in C at the Technichal University Munich [20, 5, 6, 15], and then subsequently in Pascal at the German Armed Forces University Munich [21]. Ideas from both prototypes and lessons learned from their use brought forth the development of the present version [10, 23, 26], implemented in Smalltalk-80.

HOPS is a program transformation system for a functional language centered around a very simple kernel language along the lines of typed λ-calculus. The intention of this extremely basic foundation is to achieve closeness to abstract semantics; in [27] a relational semantics for this kernel could be formulated.

HOPS implements strong typing; the typing system is a derivate of a typing system after Church (and not after Curry like the Hindley-Milner type systems in use in most functional languages). Nevertheless, whilst there is generic typing, we do not mix levels into a true second-order system.

[1]partially supported by the DFG

The programs of HOPS are directed acyclic term graphs (term DAGs) and not strings or conventional terms as is usual. This is not only the internal representation as e.g. in LAMBDALG (see [12]), but is also the model with which the user interacts. The interaction is via a modern graphical user interface and allows direct graphical manipulation and transformation of HOPS DAGs.

Programming in HOPS lends itself more naturally to a functional style (see [16]) than programming in conventional functional languages like Haskell [13].

The vision behind HOPS is that of "mouldable programs", i.e. programs that need not be static in their formulation, but can be moulded interactively into the form most corresponding to the current needs.

This paper is not intended to be a system description. The aim is to communicate at a relatively high level how design decisions of the language and of the system influence the processes of programming and software development.

2 Shortcomings of Conventional FP-Systems

2.1 Lacking Grip on Structure

LISP, a language that was one of the first to allow functional programming, enjoys the advantages of having a very simple syntax, and the resulting computer-friendly but generally human-unfriendly appearance of most LISP programs has been provoking much criticism ever since. Starting with ML, one of the design objectives of functional programming languages was to provide better readability of programs for human readers, reflecting a point of view taken most vehemently by the literate programming movement.

Following mathematical usance, functional programming languages have been provided with operators of various fixities and precedences, and more and more also with the possibility to have such operators defined by the programmer. Nevertheless, it turns out that programs written in these languages are not necessarily easier to parse for the human reader than LISP programs: the problem has been shifted, but not solved.

In our opinion, the root of the problem lies in the fact that functional programs tend to have a structure that extends more into depth and less into breadth than most imperative programs. A linear string representation of such programs is, therefore, bound to obscure much of their structure, and it requires unacceptable efforts on the part of the reader to reconstruct it.

A second source of similar problems is the typing of subterms. Apart from the LISP tradition, most functional programming languages tend to provide explicit static typing with the possibility of building program fragments with very complex types. Higher-order functions are one field of applications; another includes modern approaches to "imperative" features in functional programming, like continuations or monads for I/O and various state related aspects [24], or "fudgets" for graphical user interfaces [8], that rely on making the complex semantics of these features explicit. This complexity is then reflected in the types of the related combinators.

The programmer employing these features frequently has a hard time figuring out the right way of composing the right combinators. Errors in this respect are often hard to locate with the help of the compiler error messages and even harder to understand, once the mind is set on a certain route.

Although types are said to be "pervasive" in modern functional programming languages, as stated e.g. in the Haskell tutorial [14], they are just *not there* during the coding process, i.e., not immediately accessible by the programmer.

This lacking grasp on more complex structures has the ill effect to let programmers only hesitantly employ important powerful features of functional programming languages, thus counteracting much of the motivation behind their design.

2.2 Language Aspects

Since Backus' Turing award lecture [1] leading researchers in the field of functional programming languages have been advocating the heavy use of higher order combinators for program construction. The resulting programs should hardly contain a subterm or even a variable on the object level, that is with nonfunctional type.

Reality, however, is still different. Inspecting even very modern large functional programs reveals that the applicative style is still dominant.

We plead that this is in part due to the lack of a really close grip on program structure and typing discussed above, in part to insufficient use of truly generic data type construction.

Genericity of Haskell Data Type Constructions

A generic data type construction in Haskell is, in a nutshell, one that can be used inside a `type` statement.

Haskell supports three generic data type constructions: tupeling, function-type construction and list construction. The other two important data type constructions, viz. direct sum and data type recursion, are left to ad hoc declarations. In the Haskell setting it is still possible, however, to define a generic sum and this has been done in one extension of the standard prelude, using the name `Either`.

When browsing through the standard prelude, one notices that lists are supported extremely well, functions are still taken considerable care of while tupels hardly appear at all. Neither does the extension providing a generic sum bother to supply appropriate combinators.

The result of this state of affairs is that wherever sum data types have to be treated in Haskell programs, this tends to be done by pattern matching, introducing first-order terms and variables. The same statement holds to a lesser extent for tupels. The treatment of branching control flow by conditionals is another first-order feature.

Pointedly we might observe that the construction of list types is the only generic data type construction in Haskell truly supporting higher-order programming.

Generic Sum and Recursion

On the right we show a Haskell definition of a generic sum to be used in an extended standard prelude together with its most important combinator "`fsum`". This combinator allows translation of data variants into branching control flow by functional composition and is therefore an important help when eliminating first-order terms.

```
data a+b = Left a | Right b
fsum::  (a->c) -> (b->c) -> (a+b) -> c
fsum f g (Left a) = f a
fsum f g (Right b) = g b
```

Now we show some pseudo-Haskell code for implementing generic data type recursion "`Fix`" together with the two isomorphisms "`abstract`" and "`represent`" between the fixed point domain of a type functional "τ" and the image of that fixed point domain under "τ".

```
data (Fix τ) = Abstr (τ (Fix τ))

abstract::  (τ (Fix τ))->(Fix τ)
abstract = Abstr

represent::  (Fix τ)->(τ (Fix τ))
represent (Abstr x) = x
```

Unification for the typing of this kind of recursive data types is second-order and perhaps not even decidable in the general case — the special situation in HOPS is currently under investigation.

Nevertheless this generic approach still provides a discipline for defining recursive data types, and in standard Haskell we can use the definitions given here for lists:

```
listAbstr::  (()+(a,[a])) -> [a]       listRepr::  [a] -> (()+(a,[a]))
listAbstr (Left ()) = []               listRepr [] = Left ()
listAbstr (Right (x,xs)) = x:xs        listRepr (x:xs) = Right (x,xs)
```

To make the point more clearly, what we would like to be able to do for introducing, e.g., a tree data type, is to write

$$\texttt{type Tree a = Fix } (\lambda\ T.\ \mathbb{1} + (T \times \texttt{a} \times T))$$

and then use the generic isomorphisms "`abstract`" and "`represent`" introduced above. As long as this is impossible, or in cases where this is unwanted, we would prefer the interface consisting of

```
data Tree a

treeAbstr::  (()+(Tree a,a,Tree a)) -> Tree a
treeRepr ::  (()+(Tree a,a,Tree a)) <- Tree a
```

to the conventional one providing constructors.

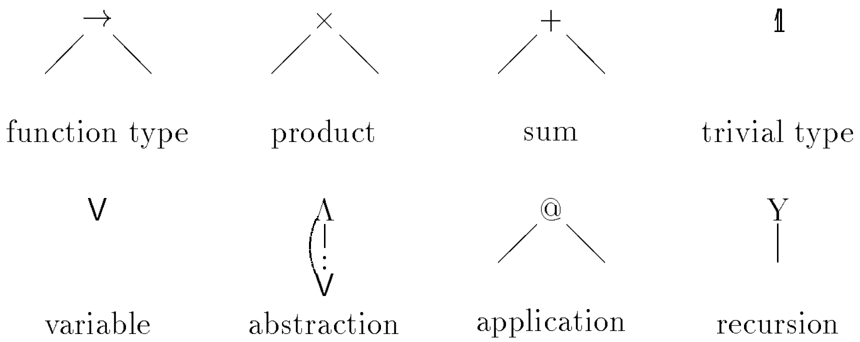
Note that the use of constructors in pattern matching (for "destruction") induces an applicative component-level programming style, while the use of the analysis function "`represent`" offers itself for higher-order functional programming.

3 HOPS Language

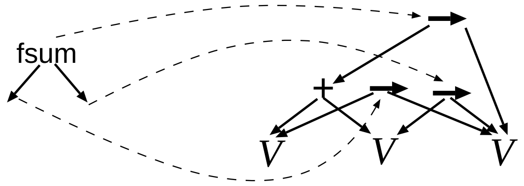
A HOPS program is one kind of term graph, typed with other term graphs. Arity and sequencing of constructors are replaced by the concept, that every constructor, i.e. every node label, has a fixed set of edge labels associated with it, such that in a HOPS DAG every node n of the given kind has for every edge label e in the aforementioned set exactly one edge labelled with e leaving n.

In presenting the language the initial direction of an edge leaving a node corresponds to its label. This means that the function application label "@" has associated with it the set of edge labels $\{\mathit{left}, \mathit{right}\}$. One of the central ideas of HOPS is to let the user interact directly with HOPS program DAGs via a graphical user interface.

So, the HOPS "language" comes in an unusual *shape*. For the *contents*, the kernel language has been designed to be close to the semantics and the generic data type constructions. Therefore, the type level of this version of the typed λ-calculus contains the following primitives:

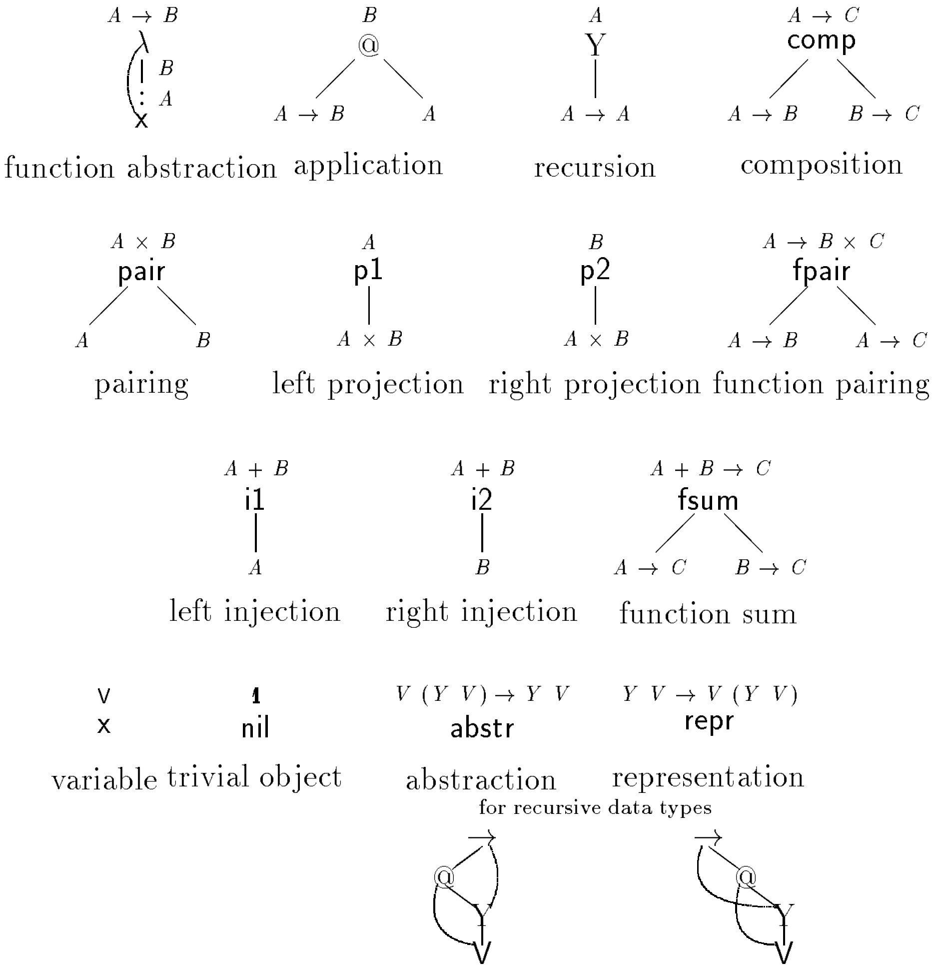


These primitives are now employed to give the typing of the primitives of the object level. The typing of an object constructor with leaving edges of labels E is defined by a type DAG together with a mapping from $\mathbb{1} + E$ to nodes of that type DAG, denoting the type of the resulting node itself depending upon the types of the successors. In the following diagram, the dashed arrows depict the said mapping and the label set of the leaving edges is $\{\mathit{left}, \mathit{right}\}$.



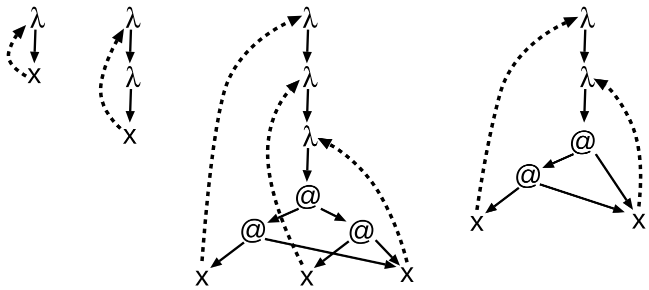
In term DAGs, variable names are superfluous as variable identity may be implied by node identity. The *binding* of variables to λ-abstraction nodes is an explicit part of a HOPS DAG and is not a derived concept as in most versions of λ-calculus.

Object level DAGs in HOPS always are attributed with corresponding type DAG nodes; the string representation chosen in the table below is easier to grasp when graphically depicting types as attributes in an object-level HOPS DAG. The free type variables then have to be given names such as A, B and C in order to be distinguishable in the string representation. In the table below, type DAGs are depicted only for data type abstraction and data type representation.



The typing algorithm involves DAG unification, which becomes higher-order in the presence of data type abstraction and data type representation. Since this may be undecidable, heuristics have been implemented for the common cases of such data type constructions, and the discipline for defining recursive data types introduced in Sect. 2.2 is supported by automatical creation of access functions for specified data type recursions.[2]

There is no need to have *variable names* for the variables V and x, since the possibilities of DAGs are exploited so as to have only one node for every required variable—variables and binding are handled the same way on the object level as on the type level.



Shown to the right as examples of HOPS DAGs are those corresponding to λ-calculus expressions for the combinators $\mathbf{I} = \lambda\, x.x$, $\mathbf{K} = \lambda\, x.\lambda\, y.x$, $\mathbf{S} = \lambda f.\lambda\, g.\lambda\, x.f\ x\ (g\ x)$ and $\mathbf{W} = \lambda f.\lambda\, x.f\ x\ x$.

[2] In the current implementation, the type level is kept consistent with respect to functional abstraction etc. by considering it as another instance of a typed λ-calculus. Type DAGs are again typed with DAGs from the level of *hypertypes*, corresponding to the kinds of, e.g., [7]. Note that, however, we do not allow direct mixing of these three levels, unlike most second-order λ-calculi.

It would not be very fruitful — and the DAGs would grow rather large — if one was restricted to programming in this small kernel language. The language can be extended by the declaration of new DAG constructors. Such declarations have to contain the name of the new constructor, the arity and the type information. For the latter, a type DAG has to be produced, and the node itself and every one of its successors must be assigned one of the nodes of that type DAG as their respective typing. If the type DAG contains free variables, the new language element is a generic one.

Extensions of the type language may be declared in the same way — the hypertype DAGs required for typing here are usually very simple.

Modules are used to organise the language.

4 HOPS Rules

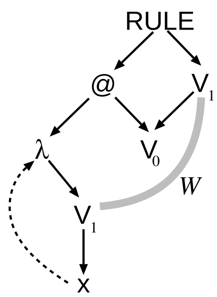
HOPS allows term DAG rules that are similar in the power of expression to the rules of combinatory reduction systems (CRSs, see [18]). Term graph rewriting in this context is an extremely complicated affair where we build upon new insights from research stimulated by HOPS [17]. Here we can only give a short exposition of a few aspects relevant to actual programming.

Combinatory reduction systems are an extension of term rewriting systems (TRSs) covering rules, where substitution is part of the rule definition, as e.g. in the β-reduction. This is achieved by introducing *metavariables* with arity zero or greater instead of the TRS variables, that always have arity zero. Additionally a concept of variable binding is considered with bindable variables and binding constructors being part of the ground language.

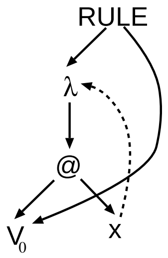
The number of successors of rule parameters in HOPS (corresponding to the metavariables in CRSs) may be either zero or one (in CRSs it is arbitrary), but the restriction existent in CRSs, that successors of the metavariables must be bound (object) variables, can be significantly weakened to the condition that every metavariable with one successor has at least one *rigidly encapsulated* occurrence in the left hand side. This means roughly that it must lie on a path between a binding node and its bound variable, where no other metavariable node relies on the same path for its rigid encapsulation.

A rule itself is a term graph with two entries, the sources of the left and right hand sides respectively. These are drawn as the left and right successors of a fictitious node marked RULE.

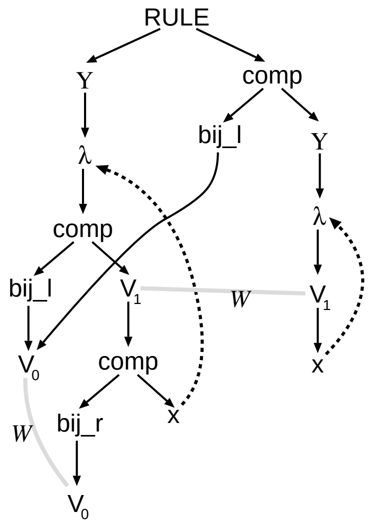
For rules, the HOPS language is extended by the two kinds of metavariables, V_0 and V_1 having either no successor or one successor respectively — which are typed with independent type variables. (An extension to metavariables with more successors is under consideration.) A metavariable — most prominently one with successors — may consist of several nodes. These nodes are connected by the relation W of *variable identity*, which is part of HOPS DAGs, just like the variable binding. This way problems with α-conversion are again avoided.



The λ-calculus rule β-reduction is a CRS rule with one one-successor metavariable consisting of two nodes and is therefore a legal rule in HOPS, shown in the figure above, where the argument metavariable is shared between the two rule sides. The two nodes of the one-successor metavariable V_1 are connected by the variable identity W. The η-rule is of course much simpler, and is shown to the right: The condition, that x may not occur free in V_0, is guaranteed here by the underlying definition of the rule mechanism.



A HOPS rule that is not a CRS rule is shown in the picture to the right (employing language extensions "bij_l" and "bij_r"). Here the left hand side occurrence of V_1 not only does not have a bound variable as its successor, we also employed another extension relative to CRSs, that is the possibility of having non-left-linear rules, resulting in a DAG comparison when matching.



Language extensions may be specified or defined using rules, and rules also enter modules. There is no distinction between definition rules and specification rules, which results in a module concept very similar to LAMBDALG [12] (or Haskell within its means).

5 Notes on Programming in HOPS

Programming in HOPS has a slightly unusual feeling, and resembles working with a syntax directed editor — but there is more to it. The hands-on graphical interaction offers direct contact to the syntactical structure of the manipulated programs. No fixity and associativity tables have to be remembered; mental parsing problems belong to the past. Building DAGs by graphical interaction assures not only correct syntax; since types are always kept alongside the programs, it is impossible to build untypable programs. The omnipresent typing furthermore guides the programmer in the process of program construction — with more experience, this latter feature grows to be even more useful than the former.

The equipment with primitive data types in HOPS differs from the usual in that there are no Boolean values provided, and there is no conditional either. On the other hand, HOPS provides the direct sum as primitive data type constructor, which is quite unusual.

Variant data types in HOPS are handled using sum types, and functions dealing with them can be composed using the function sum. A standard module corresponding to standard preludes in textual functional languages provides not only many auxiliary functions to use with direct products, but also their duals for use with direct sums. After an initial phase of accustomization, the appropriate programming style results in a much more natural use of variant data types and also will appear to be more intuitive to the programmer.

Another difference lies in the treatment of recursive data types. The generic data type abstraction and representation functions of HOPS (a complete novelty introduced in HOPS for the first time) correspond to the isomorphisms between the fixed point domain of a type functional and the result of applying this functional to the fixed point domain, as these two should not be regarded as *equal*, but only as *isomorphic*. This categorical distinction is reflected in different uses of abstracted objects of recursive data types and their representation within a program, and the explicit use of the isomorphisms allows acyclic finite type graphs.

Below this level of recursion a datum of a recursive type usually reveals a variant structure, which in HOPS is handled by the direct sum. In Haskell, the individual declaration of recursive data types is dealt with by pattern matching when defining functions on these types. Here — together with the conditional — lies the main reason why so many Haskell programs never overcome the applicative style.

Programming in HOPS is never limited to writing down code or building of code graphs. The transformation capabilities of HOPS can be employed at any stage of program development. This brings along more flexibility and freedom with coding choices, so one might e.g. first construct a quick-and-ugly applicative solution and then transform this into a more pleasing functional form, or even experiment with different forms to get a feeling for the formal properties of the program in question and finally settle on a formulation employing the level of abstraction appearing to be most satisfying in the individual case.

The HOPS approach to variants and recursive data types works together with the omnipresent transformation capabilities to make a *truly functional* programming style which comes out much more naturally in HOPS than in conventional functional languages.

6 Notes on the DAG concept

The way of disposing of α-reductions has been mentioned; this is now common in the world of term graph rewriting as far as free variables — corresponding to metavariables without successors in our setting — are concerned. By considering the variable binding function as a constituent in its own right of a HOPS DAG, the effect of getting rid of α-conversion can be transferred to bound variables as well. This is also far more intuitive than for example working with de Bruijn indices and it makes the important notion — variable binding — a primitive notion instead of a derived one as usual.

HOPS DAGs do not have to be maximally identified — identification and separation are explicit transformations (that are not always allowed in non-deterministic contexts). The user has full control over the “graininess” of the DAG structure, which can be extremely important for some transformation strategies.

By developing an appropriate notion of term graph rewriting with bound variables, a rule mechanism has been implemented realizing the power of combinatory reduction systems on term DAGs, thus avoiding the restriction and somewhat unnatural dichotomy of for example LAMBDALG [12], where there are conventional term rewriting rules (implemented on DAGs) and β-reduction, as proven correct for term DAGs by Wadsworth in [25]. The DAG rewriting mechanism of HOPS applied to the β-rule as stated above also implements exactly Wadsworth's graph reduction, but other CRS rules are equally admissible in HOPS.

7 Notes on Transformations

The transformation system of HOPS is fully integrated into the programming system. This means that transformations of any complexity can be employed at any stage of the programming or development process. The simplest transformation is the application of a single rule. HOPS is capable of selecting all rules that can be applied at the selected node. When reverse application is asked for, HOPS tries only rules it knows to be invertible — the β-reduction for example is not invertible, since the metavariable V_1 is not rigidly encapsulated in the inverted rule.

Derivation and construction of new rules is also possible at any stage, the same holds for language extensions. New rules are immediately applicable when the module to which they have been assigned is activated.

Just as predefined single rules may be applied, the same holds for predefined transformation programs. The behaviour of these may be fine-tuned by setting switches or activating different sets of modules. Some may be limited to very specific tasks and always work with the same rules; others may only define a certain strategy the specific behaviour of which widely depends on the set of active rules.

Examples for such transformation programs are:

- normal order reduction, with adaptions for the application of functional sums and similar cases (a special case of the rewriting control scheme proposed by Staples in [22]),
- reversion of β-reduction after selection of two nodes — also recognizes illegal cases,
- modifying another strategy by exploiting commutativity and/or associativity of given lists of patterns,
- automatic construction of code pieces from types such as catamorphisms (see [9]).

Besides such predefined programs, the user may at any time add programs of his own design, where existing ones can be built upon. HOPS provides a large array of tools for transformation programming, such as a programming language interface with special primitives or several DAG attribution mechanisms.

The prominent place of normal order reduction in the list above stems only from the fact that it is well known and conceptually simple. Evaluation in HOPS is always just a special case of transformation, so that there has been no attempt to invest dedicated effort into writing an interpreter to speed up "execution" of programs. Even partial evaluation, although far more important to the aims of HOPS, still has to be considered as a special case.

Transformation in HOPS is usually aimed at transforming given programs into "better" ones. This might mean e.g.:

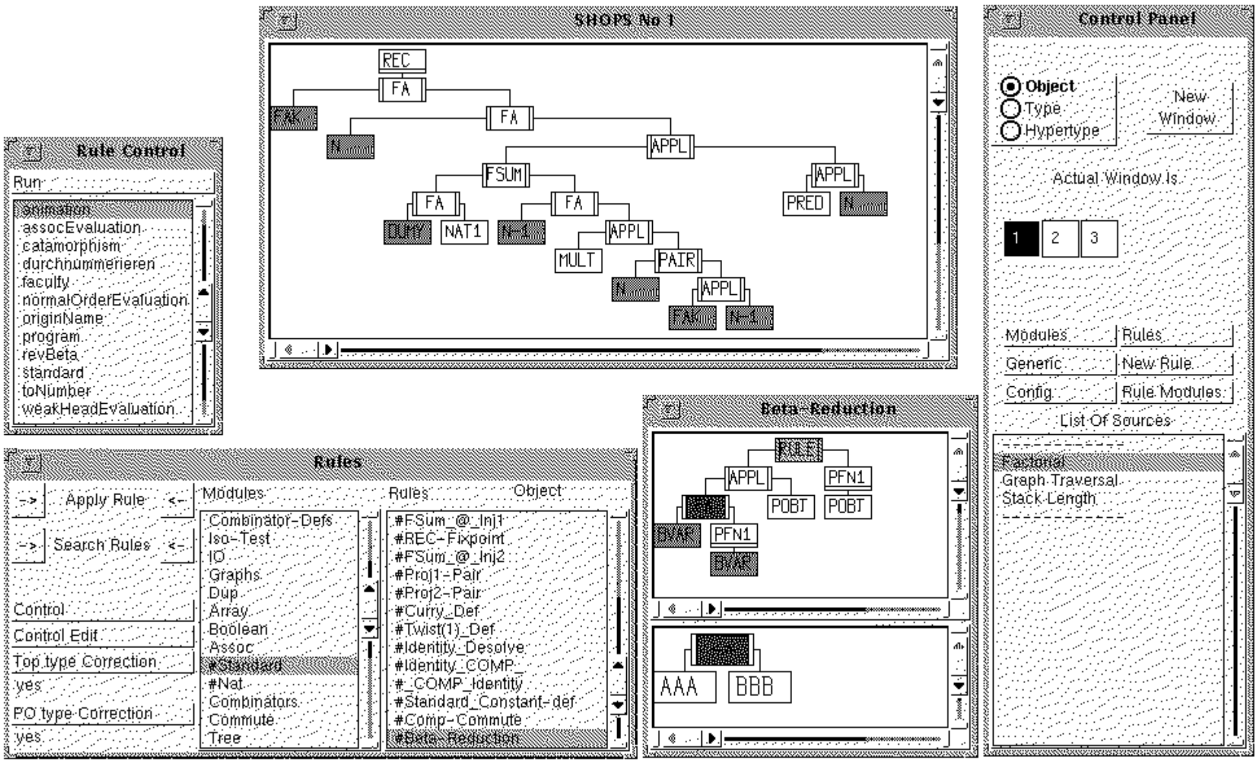
- to transform an applicative form into a functional one,
- to transform an expensive recursion (modulo some execution model) into a cheaper one,
- to transform high level code into a form reflecting a given machine architecture or even
- *reverse engineering*: transform low level code into a higher level of abstraction.

As a keynote there is the field of *code generation*. Some of the transformations hinted at above already are aimed in this direction; other important subtargets HOPS has allowed to approach are:

- automatical conversion into a "functional normal form",
- syntactically guided introduction of exception handling,
- transformation onto abstract machines, e.g. sequential memory machines.

8 Development Environment

For giving at least a superficial impression of the development environment of HOPS, we show a more or less typical S-HOPS screen. To the right, the "Control Panel" is at the heart of interacting with the different aspects of HOPS administration. To its left, a DAG view can be seen — windows like this one are used for direct graphical interaction with program DAGs and type DAGs. It is possible to simultaneously open up to twelve DAG views, and they can individually directed at one of the three layers — these functions are accessible from the top part of the control panel. The bottom part of the panel displays a list of the sources of the level the actual DAG view window is looking into — that window displays all the nodes reachable from the selected source. The induced rooted subDAG can be scrolled, and the display scale can be chosen from three different possibilities.



The array of buttons in the middle of the control panel gives access to different aspects of HOPS module administration, and via the "Rules" button to the central transformation interface window in the lower left corner. Here rule sets originating from the module organisation can be activated, deactivated and inspected — single rules can be shown in the window in the lower right corner. A single selected rule can be applied — at the node selected in the actual

DAG view — in either direction via the "Apply Rule" buttons, and the analogous "Search Rule" buttons allow to choose from all active rules that match at the selected node. Finer points of interaction between the rule mechanism and typing can be influenced through the "type Correction" buttons, and "Control" refers to the repertoire of transformation programs listed in the top left window.

9 Program Construction Example

9.1 The Task: A Simple HTTP Demon

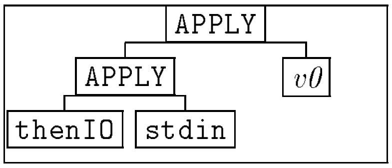
As a simple example that yet is not devoid of practical usefulness we are going to construct a simple server for the Hypertext Transfer Protocol "HTTP" that is employed by the popular distributed information system initiative World Wide Web "WWW".

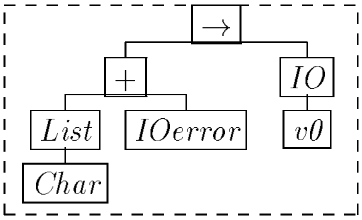
Such a server is called from the internet demon process and on the standard input channel receives a request of the form "`GET` ⟨*name*⟩ ⟨*version*⟩ ...". A server designed as a file server might then simply respond (on the standard output channel) with the contents of the file designated by *name* prefixed with a constant protocol string.

Since we believe in error handling (and since it makes the types more complicated), we are employing monadic I/O primitives corresponding to the "`...IOE`" set provided by the Glasgow Haskell Compiler, but we replaced the type `IOResult` with the corresponding instance of the generic sum, resulting in "`...IOX`" functions in the generated Haskell.

9.2 Stepwise Program Construction

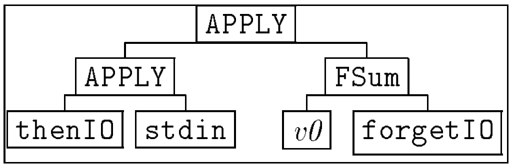
The first step is to read from the standard input. In preparation for further interaction with the world we supplied the "`readChanIOX stdin`" action as first argument to "`thenIO`". The type of the second argument, the free variable "`v0`" is now:



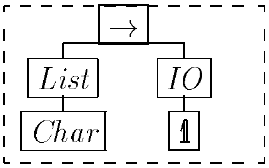


In Haskell this corresponds to "`(String+IOError) -> IO v0`". So we should supply a function that can cope with a direct sum as argument type and return and IO-action. Which type this IO-action should contain as return type is still unspecified. But since we are going to write a standalone program, this will have to be "`IO ()`".

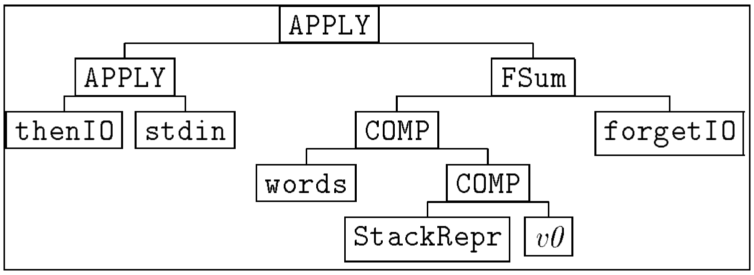
We now supply an "`fsum`", and for the second branch, which is the error branch, we enter the function "`forgetIO`" that is the constant function returning the trivial IO-action.



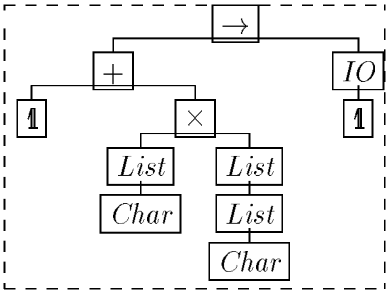
The type of the first branch has now been specialised to "`String -> IO ()`". The syntax of the HTTP request explained above suggests the use of the Haskell prelude function "`words`" for analysing the input stream.

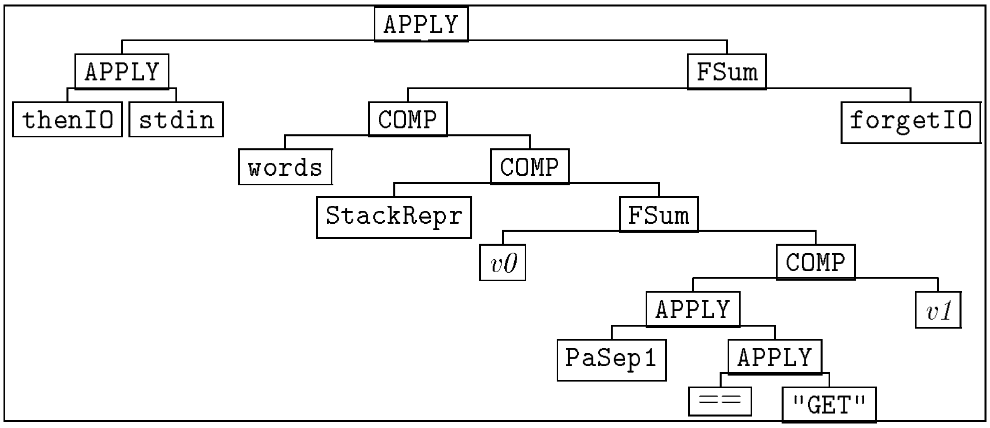


The generic treatment of recursive data types has the characteristics of not providing destructor functions like “`head`” and “`tail`”, but completely relying on the data type representation function together with direct sum and product processing for accessing the constituents. This has the advantage of avoiding partial functions together with the associated proof obligations.

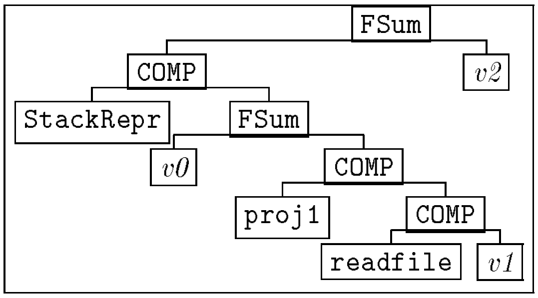


The type of the function we now have to fill in for the free variable looks rather complex; in Haskell it would be “`(()+([Char] , [[Char]])) -> IO ()`”. But by concentrating on the top-level structure — and this is much easier in HOPS than in Haskell — we immediately see that the next node should be an “`fsum`”. We will not bother about the syntax error lying behind the first branch at the moment, and therefore leave the free variable there untouched. The product “`(String,[String])`” of the second alternative deserves closer inspection. The first component is the first word of the input stream, while the second component is the list of the remaining words. The first word is only needed to determine the kind of request, i.e. in our special case to check whether it equals “GET” or not. Afterwards, it may be forgotten about. Therefore we employ a special combinator “`PaSep1::(a->(b+c)) -> (a,d) -> (d+d)`” which is a very close analog to conditionals that otherwise are not supported in HOPS.

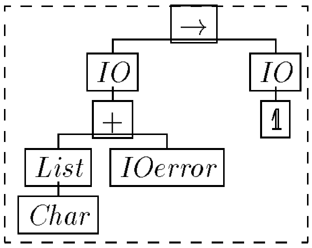




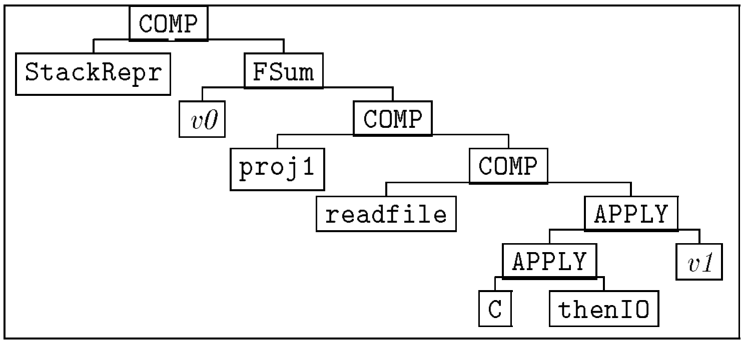
The free variable “`v1`” above has the type “`([String]+[String]) -> IO ()`” and will be instantiated with the graph at the right. We again concentrate on the non-error branch. The list of remaining words has to be analysed, and if it is nonempty, the first one of these words — which is the second word of the input stream — is all we want to keep for further processing. Hence we use the projection to the first pair component to get rid of the rest. We then use that word to perform a “`readFileIOX`” operation.



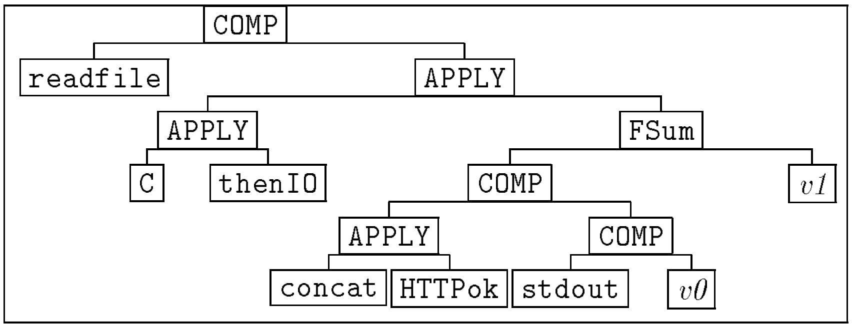
The type of the open node now has a top-level structure we have not encountered before. By comparison with the type of the combinator necessary here intuitionally, "`thenIO`", we notice that the type of the latter comprises an additional second argument. What we need here, is therefore "`thenIO`" with switched arguments, which can be obtained by applying the C combinator from λ-calculus.

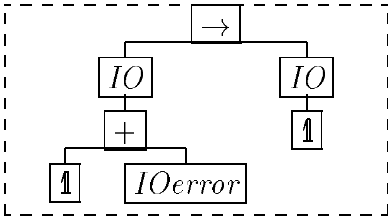


Now the open variable "`v1`" has a type we can cope with: "`([Char]+IOError)->IO()`". The list of characters here is the contents of the file we just read in. We prepend the protocol header to that string and then write the whole result to the standard output channel.



What remains to be done is only finishing off. The type of "`v0`" (see below) contains an "`IOError`", we can safely ignore. Otherwise we might append it to some log file.





The IO-primitive for ignoring errors in situations like this is "`finishIO`". Both "`finishIO`" and "`forgetIO`" have been defined in HOPS and automatically exported to Haskell as:

```
finishIO = combC (thenIO_) (returnIO ())
forgetIO = (\bvar -> returnIO ())
```

The only first-order nodes of the resulting program — with reasonably filled in error handling — turn out to be the protocol strings. A direct translation into Haskell yields:

```
module Main ( mainIO ) where
import PreludeGlaIO
import STDX

mainIO = let m1 = (finishIO . appendChanIOX "stdout") in
         let m2 = (m1 ("HTTP/1.0 400 syntax error\n")) in
         let m3 = (\x-> m2) in
     thenIO (readChanIOX "stdin")
     (fsum (((fsum m3 ((fsum ((fsum m3 (((combC (thenIO)
          (fsum ((m1 . (++) ("HTTP/1.0 200 ok\nContent-type: text/html\n\n")))
        ((\x-> m1 ("HTTP/1.0 500 internal error\n")))) . readFileIOX) . fst))
       . listRepr)) (\x->m2) . paSep1 (equalB ("GET")))) . listRepr) . words))
     (forgetIO))
```

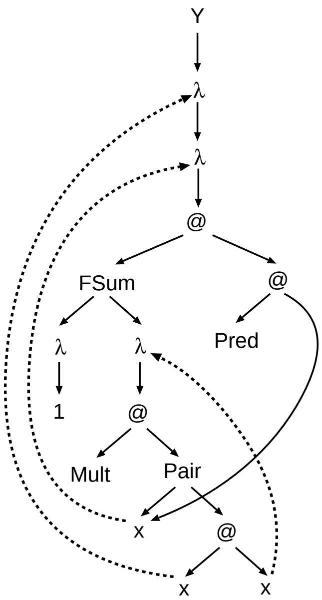
The "`let`" constructions stem from DAG sharing not visible in the simple representation of HOPS screens we employed above.

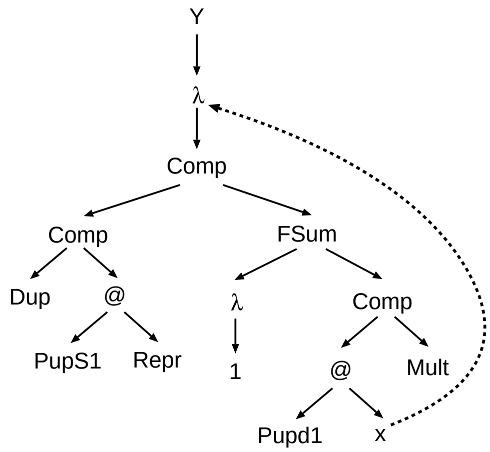
This Haskell program (without the random layout introduced for printability) has been compiled using the Glasgow Haskell Compiler and has been successfully tested as an internet demon sending it requests from the WWW-client "Mosaic" on a remote machine.

10 Program Transformation Example

In the figure below to the left, a version of the factorial function is shown. Of its seventeen nodes, only nine denote functions. This, together with the many first-order abstractions, indicates that this version is written in an applicative style.

Within HOPS, employing the repertoire of primitives and laws for direct sums, direct products etc., it is not hard to derive a functional version, depicted to the right. For reasons of space, we have left out the transformation steps and we only show the result — of its sixteen nodes, fifteen denote functions. Only the constant "1" has a component-level type. ("Repr" is the representation function for natural numbers, typed $\mathbb{N} \to \mathbb{1} + \mathbb{N}$, that simultaneously acts as a zero-test and, if possible, provides the predecessor.)





Here, "Dup" is a diagonalisation function, "Pupd1" lifts its argument function into the first component of a direct product, and "PupS1" does the same, followed by distributing the direct sum (that has to be the result type of the argument function) over the product.

This functional representation obviously is a linear recursion, but not a repetitive one. For transformation into repetitive form, we may use the fact that multiplication is associative. In more conventional notation, a transformation rule for this purpose is given in [4, p. 283]:

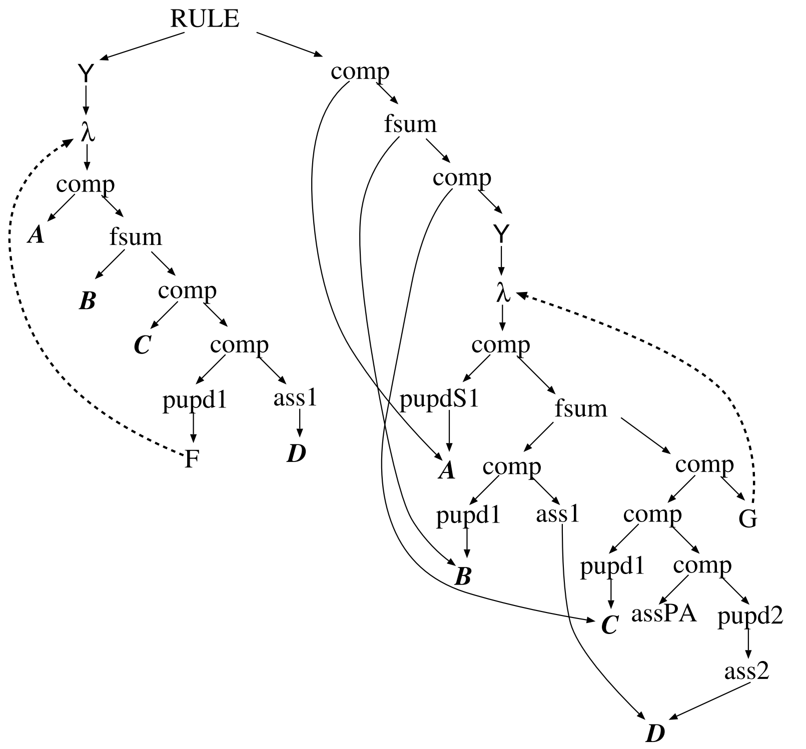
$$
\begin{array}{llll}
\textbf{funct} & L \equiv (\lambda\ \ m)\ \rho : & & \\
 & \textbf{if } \mathcal{B}[m] & \textbf{then} & \phi(L(\mathcal{K}[m]), \mathcal{E}[m]) \\
 & & \textbf{else} & \mathcal{H}[m] \qquad\qquad \textbf{fi}
\end{array}
$$

$$
\updownarrow \qquad \left\{ \begin{array}{l} \psi : \nu \times \nu \to \nu : \\ \phi(\phi(r, s), t) = \phi(r, \psi(s, t)) \end{array} \right.
$$

$$
\begin{array}{llll}
\textbf{funct} & L \equiv (\lambda\ \ m)\ \rho : & & \\
\lceil & \textbf{if } \mathcal{B}[m] & \textbf{then} & G(\mathcal{K}[m], \mathcal{E}[m]) \\
 & & \textbf{else} & \mathcal{H}[m] \qquad\qquad \textbf{fi where} \\
 & \textbf{funct} & G \equiv (\lambda\ \ m, \nu\ z)\ \rho : & \\
 & \textbf{if } \mathcal{B}[m] & \textbf{then} & G(\mathcal{K}[m]), \psi(\mathcal{E}[m], z)) \\
 & & \textbf{else} & \phi(\mathcal{H}[m], z) \qquad\qquad \textbf{fi} \rfloor
\end{array}
$$

One of the draw-backs of this formulation is its ad-hoc treatment of what corresponds in CIP [2, 3] to the metavariables of CRSs [18]. Also, the rule is aimed at applicative programs.

In HOPS, we give a functional style formulation of this rule, therefore not needing unary metavariables. Also, two of the parameters of the rule above ($\mathcal{K}[\,]$, $\mathcal{E}[\,]$) collaps into one ($\mathbf{C}$) in the HOPS rule. (The unary metavariables are named with letters for simplifying the drawing.)



Obviously, all that is necessary for applying this rule to the functional version of the factorial, is to introduce a composition with an identity function for $\mathbf{C}$, and an introduction of the knowledge about associativity of multiplication, that is coded into an abstract data type with projections "ass1" and "ass2". In the repetitive version of the factorial, which could thus be arrived at, a few simplifications would be possible, reducing the result to a shape that directly corresponds to efficient implementations — indeed, low-level code could be emitted from this stage. For the sake of brevity, we have to sacrifice the details here.

11 Outlook and Conclusion

The graphical term DAG manipulation and transformation system HOPS enters the stage of program development systems equipped with an array of novel features. DAGs at the user interface, language elements hardly above the level of denotational semantics, term DAG rewriting with the power of combinatory reduction systems, and a whole array of transformation programming facilities form a unique environment. This environment requires a little training, but the flexibility and power of expression provided make the effort worth while.

This paper concentrates on the surface of the system; many interesting problems that have at most been hinted at invite further research. HOPS as it stands already is an extremely versatile tool for employing several novel ideas in formal software development.

We are in the course of extending the module concept to add most of the power of Haskell's — excluding import cycles, since we want to obtain a manageable DAG structure here as well — and to comprise elements of literate programming and hypertext.

Furthermore we plan to implement a genuine DAG representation on the screen and the possibility to switch to higher-order dataflow graph representations (see e.g. [11]) for at least certain shapes of HOPS programs.

Finally we want to gratefully acknowledge the opportunity to work at and with HOPS given to us by Gunther Schmidt.

References

[1] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *LNCS*. Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[3] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Volume II: The Transformation System CIP-S*, volume 292 of *LNCS*. Springer-Verlag, Berlin, 1987.

[4] Friedrich L. Bauer and Hans Wössner. *Algorithmische Sprache und Programmentwicklung*. Springer-Verlag, 2. edition, 1984.

[5] Ludwig J. Bayer. Entwicklung von komplexen Werkzeugen zur interaktiven Auswertung von DAGs im Programmiersystem HOPS. Diplomarbeit, TU München, 1987.

[6] Ludwig J. Bayer, Rudolf Berghammer, and Peter Kempf. Programmieren mit höheren Objekten: Bausteine und Regeln im HOPS-System. Technical Report 9003, Universität der Bundeswehr München, Fakultät für Informatik, May 1990.

[7] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, 1990.

[8] Magnus Carlsson and Thomas Hallgren. FUDGETS, a graphical user interface in a lazy functional language. (draft, Jan 1993, contained in the Chalmers University Haskell Distribution).

[9] Oege de Moor and Richard Bird. Solving optimisation problems with catamorphisms. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *2nd Conference on the Mathematics of Program Construction*, volume 669 of *LNCS*, pages 45–66, 1992.

[10] Klaus Endres and Frank Müller. Objektorientierte Implementierung eines Programmiersystems mit zweischichtig typisierter DAG-Sprache und Regelanwendung. Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, December 1991.

[11] Alex Fukunaga, Wolfgang Pree, and Takayuki Dan Kimura. Funktions as data objects in a data flow based visual language. In Kwasny and Buck [19].

[12] Yexuan Gui and Mitsuhiro Okada. LAMBDALG: Higher order algebraic specification language. In Claude Kirchner, editor, *Rewriting Techniques and Applications: 5th international conference, RTA 5, Montreal, Canada, June 16–18, 1993*, volume 690 of *LNCS*, pages 462–467, Berlin, 1993. Springer.

[13] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[14] Paul R. Hudak and J.H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):1–53, 1992.

[15] Wolfram Kahl. *HOPS — Higher Object Programming System, Functional Graphics-Based Interactive Programming, User Manual.* Fakultät für Informatik, Universität der Bundeswehr München, January 1991. (internal manual).

[16] Wolfram Kahl. Can functional programming be liberated from the applicative style? In Bjørn Pehrson and Imre Simon, editors, *Technology and Foundations, Information Processing '94, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August – 2 September 1994, Volume I*, volume A-51 of *IFIP Transactions*, pages 330–335. IFIP, North-Holland, 1994.

[17] Wolfram Kahl. Algebraic term graph rewriting with bound variables. In *Proc. Fifth International Workshop on Graph Grammars and Applications to Computer Science, Williamsburg, Virginia, 13–18 November, 1994*, LNCS. Springer Verlag, 1995. to appear.

[18] Jan Willem Klop. Combinatory reduction systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980. PhDThesis.

[19] Stan C. Kwasny and John F. Buck, editors. *1993 ACM Computer Science Conference, February 16–18, 1993, Indianapolis.* ACM press, 1993.

[20] Klaus Libera. DAG-Transformationen für benutzerdefinierte Typen im graphisch gestützten Dialog. Diplomarbeit, TU München, 1987.

[21] Gunther Schmidt, Ludwig J. Bayer, and Peter Kempf. Hinweise zur interaktiven Konstruktion von Bereichen, Objekten und Programmtexten im HOPS-System. Technical Report 9004, Universität der Bundeswehr München, Fakultät für Informatik, May 1990.

[22] John Staples. Optimal evaluations of graph-like expressions. *Theoretical Computer Science*, 10:297–316, 1980.

[23] Jörg Stender. Erweiterung des DAG-orientierten Programmiersystems Smalltalk-HOPS um Modulverwaltung und Transformationssystem. Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, 1992. ID 42/92.

[24] Philip Wadler. The essence of functional programming. In *19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. acm press. invited talk.

[25] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* D.phil. thesis, Oxford University, September 1971.

[26] Mathias Weitz. Eine Schnittstelle zum Erstellen komplexer Transformations-Strategien in Smalltalk-HOPS. Diplomarbeit, Universität der Bundeswehr München, Fakultät für Informatik, 1993. ID 28/93.

[27] Hans Zierer. *Programmierung mit Funktionsobjekten: Konstruktive Erzeugung semantischer Bereiche und Anwendung auf die partielle Auswertung.* Dissertation, Technische Universität München, Fakultät für Informatik, 1988. Report TUM-I8803.