The Term Graph Programming System HOPS

Wolfram Kahl

1 Introduction

Programmers of Haskell or ML know that once their programs are syntactically correct and get past the type checker, the probability that they are correct is usually already pretty high. But while the parser or the type checker still complain, it is not always easy to spot the real problem, which may be far from the trouble spot indicated by the often cryptic error message.

Syntax-directed editing can solve the problem with syntax errors. But, even if a syntax-directed editor would be extended with online typing, another problem would be how to present the typing information to the user; huge type expressions with unclear relation to the program expression may perhaps be of only very little help.

We present a system that deals with both problems by making the program structure *and* the program's typing structure explicit and interactively accessible in the shape of term graphs. The design of this system relies on recent advances in the theories of untyped and typed second-order term graphs.

The **H**igher **O**bject **P**rogramming **S**ystem HOPS, which has been developed by a group led by Gunther Schmidt since the mid-eighties (Bayer et al. 1996, Kahl 1994, Kahl 1998, Zierer et al. 1986), is a graphically interactive term graph programming system designed for transformational program development.

In HOPS, only syntactically correct and well-typed programs can be constructed. The choice of the language is only constrained by certain restrictions of the term graph formalism and of the typing system. An important ingredient of HOPS is the transformation support; this favours referentially transparent languages — for this reason most of our examples use a language close to typed λ-calculi and Haskell. HOPS is intended as:

Research tool: New languages can easily be constructed and experimented with via the transformation mechanism.

Programming tool: HOPS may be used as a "better Haskell editor", integrating syntax directed editing with strong online typing. The editing safety more than weighs up the time required to get used to the term graph view.

Visualisation and debugging tool: Automatic evaluation sequences help to illustrate the workings of, for example, purely functional programs with lazy evaluation — in the absence of accessible Haskell debuggers or tracers this is especially useful.

Education tool: A hands-on experience of term graphs, syntactic structure, typing, graph reduction, general second-order rule application and reduction strategies can be gained easily through the use of HOPS.

The current version (available via WWW) features several significant improvements

over earlier prototypes:
- The theory behind the term graph transformations is now well-understood,
- typing has been integrated into the term graphs both theoretically and in the implementation,
- HOPS is now an essentially language-independent framework, and
- literate programming ideas are more adequately integrated.

All term graph drawings in this paper have been generated from the latest version of HOPS.

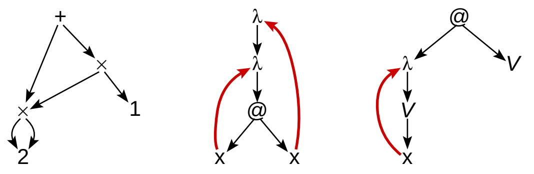
2 Principles of HOPS

HOPS manipulates arbitrary second-order term graphs, where all the structure usually encoded via name and scope is made explicit.

Term graphs in HOPS therefore feature nameless variables, explicit variable binding (to denote which node binds which variable), explicit variable identity (to denote which nodes stand for the same variable) and metavariables with arbitrary arity.

All the problems usually connected with name clashes and variable renaming are therefore avoided.

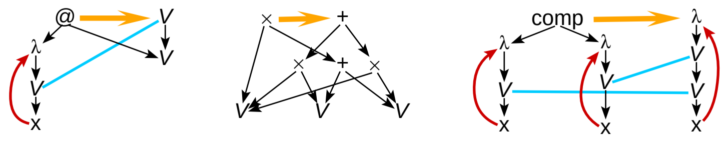
Consider the following examples, where successor edges are black arrows with their sequence indicated by the left-to-right order of their attachment to their source node; binding edges are drawn as thick, usually curved arrows and an irreflexive kernel of variable identity is indicated by medium thick simple lines:



The first two correspond to the terms "$2 \cdot 2 + 2 \cdot 2 \cdot 1$" from arithmetic and "$\lambda x.\lambda f.f\,x$" from λ-calculus — notice that in the linear notation the two different bound variables need different names "f" and "x" since they occur in the same scope, while in the term graph different nodes labeled "x" always represent different bindable variables.

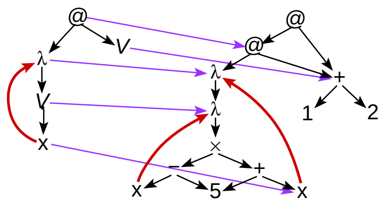
For the last two term graphs, let us assume that A and B are metavariables — in HOPS, arity is part of the metavariable node label, so that unary and zero-ary metavariables all are drawn with the label V, but according to their arity they should be considered as different labels V_0, V_1 in the examples. The last two term graphs then correspond to the metaterms "$(\lambda\, x.\, B[x])A$" and "$B[\mathsf{rec}\ x.\, B[x]]$", respectively.

At the heart of HOPS usage is term graph transformation; **rules** are also given as term graphs, where an additional thick long-tipped arrow connects the roots of the left- and right-hand sides. As examples see the rule "$(\lambda\, x.\, B[x])A \;\longrightarrow\; B[A]$" for β-reduction in λ-calculus, the rule "$x \cdot (y + z) \;\longrightarrow\; x \cdot y + x \cdot z$" for distributivity of multiplication over addition, and a rule "$(\lambda\, x.\, B[x]) \circ (\lambda\, x.\, A[x]) \;\longrightarrow\; (\lambda\, x.\, B[A[x]])$" for the composition of two λ-abstractions (all shown with hidden typing):



Application of this kind of second-order term graph rules has been defined by Kahl (1996). The kind of homomorphisms used for matching even allows rules that are not left-linear; the resulting second-order term graph rewriting concept corresponds to an extension of the Combinatory Reduction Systems of Klop (1980), also described by Klop et al. (1993).

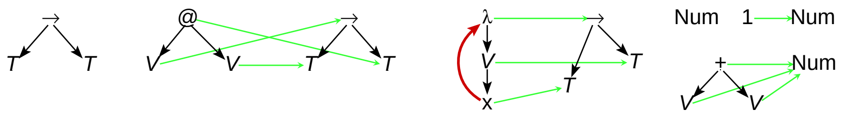
As an example homomorphism we show a β-redex, with the thin, dark grey arrows indicating the homomorphism:



Application of rules is always possible while editing term graphs in HOPS; this gives a useful flexibility to the coding process and even enhances the resulting coding style, since it is inexpensive to try several equivalent formulation before committing oneself. Besides application of single rules, the use of automatic transformation mechanisms is also possible. When all intermediate graphs of an automatic transformation sequence are displayed, a visualisation of program execution is achieved, and this feature has already proved useful for debugging complex purely functional programs.

One of the current research objectives is to define an appropriate concept of proper "transformation programming" which will allow to distill the experience of interactive transformation into strategies and tacticals for at least semi-automatic use.

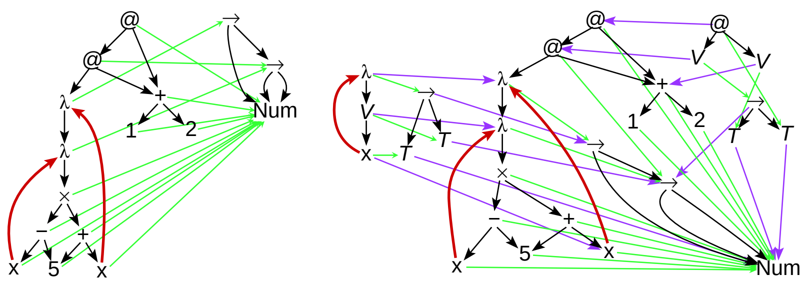
Term graphs in HOPS are strongly typed, and the **typing** is kept online all the time. The basis are *declarations* or *typing elements*, i.e., simple term graphs that introduce a new node label together with its typing schema making explicit how the typing of a node with this new label is related to the typing of its successors and bound variables. We provide six example typing elements for simply-typed λ-calculus and for arithmetics — the typing function is denoted by thin, light arrows with tiny heads, and the typing elements for $\rightarrow$ and Num, although they do not contain any typing arrows, are necessary for introducing their respective label and for fixing its arity:



Again, different nodes labeled with "*T*" are different type variables.

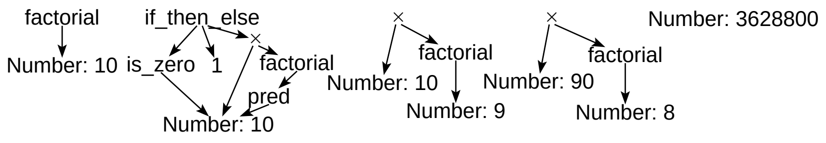
A term graph G is *well-typed* if for every node there is a homomorphism from the typing element for that node's label into G at that node; the details of this type system have been introduced by Kahl (1998). Since this system relies on the simultaneous existence of these homomorphisms to ensure well-typedness, we call this definition of well-typedness a "simultaneous" definition. As such it stands in a certain contrast to usual type systems that require an essentially *sequential* derivation or *type inference* to ensure well-typedness. This kind of type system is also very appropriate for an interactive system since it allows to operate in a very local manner for most term graph manipulations.

For graphically conveying the idea behind our definition of well-typedness, we draw a typed version of the β-redex example from above once separately (this view is available to the HOPS user) and once together with two example homomorphisms from the typing elements for function application and λ-abstraction (this admittedly rather freighted view is not available to the HOPS user).



As long as there are only zero-ary type-metavariables (labeled T) and no edges between the program and the type sides, these typing systems correspond to standard parametric polymorphism (without `let`-polymorphism). If we allow second-order metavariables on the type side, polytypic programming becomes possible, but this is not yet fully implemented in HOPS.

Data types of the implementation language, such as arbitrary-precision rational numbers or arbitrary-length strings, can be made available to HOPS programs through a special class of node labels and associated rules. The following shows a few intermediate steps of an automatic calculation of the factorial of ten:



3 User Interface

In the spirit of Literate Programming first described by Knuth (1984), HOPS modules are considered as documents addressed in the first place to humans, and these documents contain program fragments addressed also to a machine. In HOPS, code fragments are mostly declarations, i.e., typing elements, and transformation rules; so code fragments in HOPS are essentially term graphs.

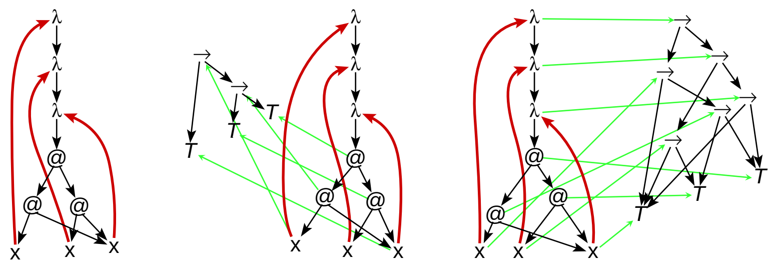
The primary interface to HOPS module documents is the HOPS **module editor**, see the picture on the next page, which presents a quasi-WYSIWYG interface to the HOPS module as a document in the literate programming style, containing text with interspersed "entries". These entries make up the different parts of the program proper and can be:

– *declarations* containing typing elements for freshly introduced language elements and also specifying the way nodes with this new label are displayed, including with special symbols (such as " $\to$ " for function types) and colour; outgoing edges also may be assigned documenting *edge labels*
– *transformation rules*, and, experimentally, *attribution rules* that introduce internal attribution edges into HOPS graphs
– *example graphs* which serve only for documentation purposes
– *Haskell attribution definitions* as described in Sect. 4
– cross references and other documentation-related special entries

The document itself may be structured in nested sections and submodules which can be folded, i.e., hidden except for their titles.

Document output of HOPS modules is possible to HTML and to PostScript via Lout (Kingston 1998) or LaTeX. HOPS modules are saved as XML documents, and future work will enhance the structuring possibilities for text, thus strengthening the capabilities of HOPS as a flexible literate term graph documentation system.

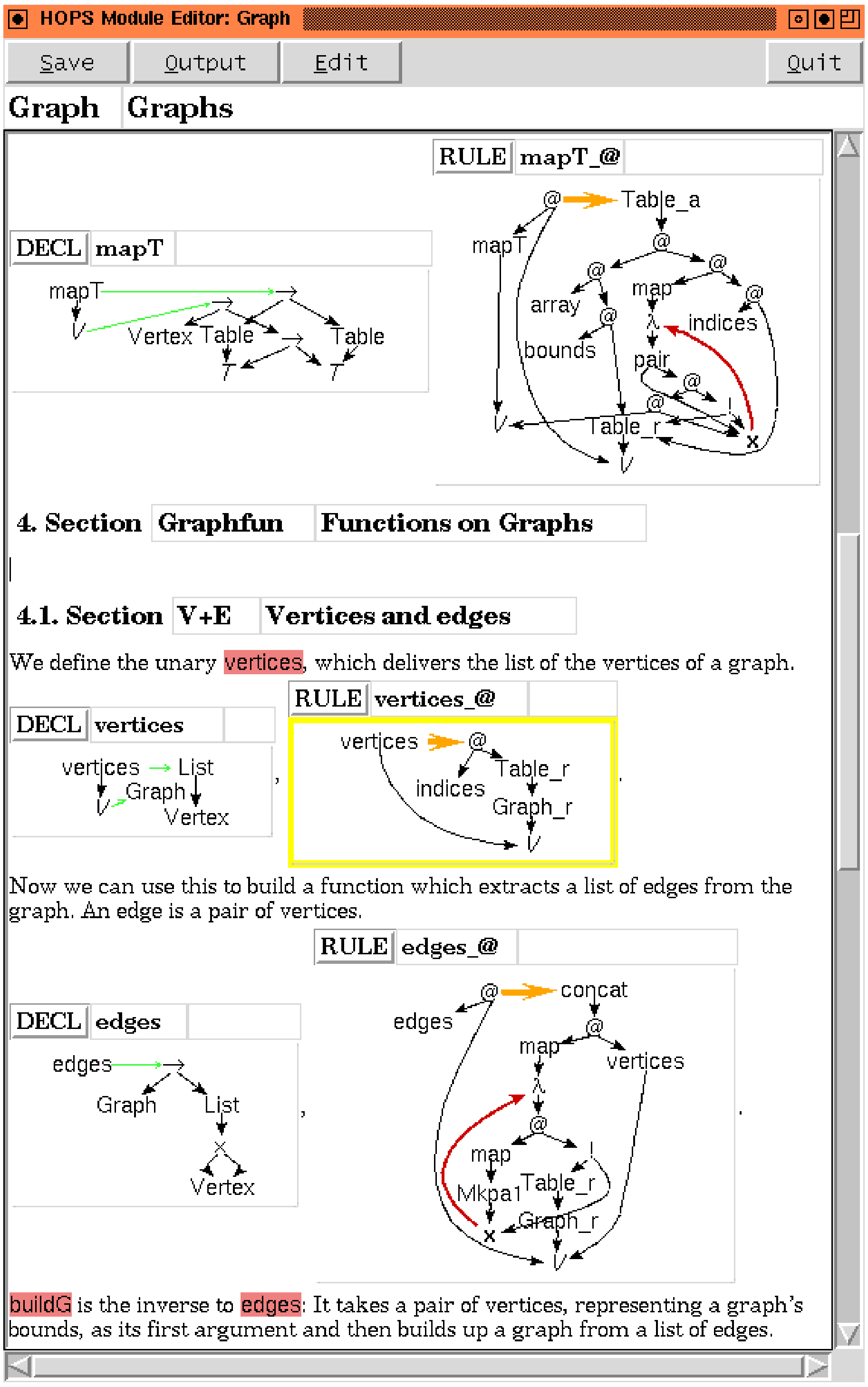
Term graph interaction takes place in term graph areas displayed directly inside the entries containing the respective term graphs, inside the text display. During program construction, most notably when combinators of complex types are involved, typing information is useful for guiding the program construction. In HOPS, this is supported by the possibility to incrementally display typing information, see e.g. the following three drawings of the S-combinator without, with some, and with full typing:



$$\lambda f.\lambda g.\lambda x.(f\ x)\ (g\ x)\ :\ (A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

Since program construction proceeds via instantiation of metavariables, turning every construction step into a homomorphism, this kind of typing display can prove very useful for finding out what types are expected at the present construction site. This instantiation of metavariables comprises the possibilities of inserting nodes with user-selected labels, regrouping, eliminating and splitting of the metavariable's successor edges, pasting selected intervals, or replacing the metavariable by other nodes.

The term graph layout is generated automatically, but is also under full manual control.


HOPS Module Editor: Graph
Save
Output
Edit
Quit
Graph
Graphs
RULE mapT_@
@ Table_a @ mapT @ @ array map @ indices @ λ bounds pair @ @ ! V Table_r x V
DECL mapT
mapT → V Vertex Table → T Table T
DECL vertices
vertices → List V Graph Vertex
RULE vertices_@
vertices @ Table_r indices Graph_r V
DECL edges
edges → Graph List × Vertex
RULE edges_@
@ concat edges @ map vertices λ @ map ! Mkpa1 Table_r x Graph_r V

4. Section **Graphfun** **Functions on Graphs**

4.1. Section **V+E** **Vertices and edges**

We define the unary vertices, which delivers the list of the vertices of a graph.

Now we can use this to build a function which extracts a list of edges from the graph. An edge is a pair of vertices.

buildG is the inverse to edges: It takes a pair of vertices, representing a graph's bounds, as its first argument and then builds up a graph from a list of edges.

4 Term Graph Attribution and Code Generation

For deriving string representations in arbitrary target languages from term graph entries or whole modules, HOPS provides a very powerful term graph attribution mechanism.

Such attributions are defined in a special kind of entries called "Haskell attribution definition". Each Haskell attribution definition consists of a target at which it is directed and the definition text proper. These definition texts take on the shape of series of Haskell definitions interspersed with "macro calls" written in a meta-syntax similar to that of FunnelWeb, a powerful literate programming system by Williams (1992). However, Haskell here serves only as the metalanguage or *attribute definition language* used to produce the real result values, which will usually be strings or functions delivering strings, and these strings may then be interpreted in the target language.

In fact, the dependence of this attribution mechanism on Haskell as its attribute definition language is very weak; only a few predefined macros employ a small fragment of Haskell syntax; the only other dependency on Haskell is the call to `hugs` as evaluation mechanism. Therefore, adapting this mechanism to a different attribute definition language would be extremely easy.

A first application of this mechanism serves to translate HOPS modules to Haskell modules; this allows HOPS to be used as a kind of "better Haskell editor" with on-line type checking and transformation possibilities; a second application will be seen below in Sect. 5 where we generate Ada code.

To give a flavour of how this attribution mechanism is used, we show here the definitions of two attributes *expr* and *pexpr* in a simplified Haskell conversion that does not respect sharing in any way and takes care of parenthesisation in only a rather crude way — the value of the attribute *expr* at a node *n* is a string containing a Haskell expression corresponding to the subgraph induced by the node *n*, and this Haskell expression is not parenthesised on the outside if easily avoidable; *pexpr* has parentheses added at least if this makes a difference. The definition for the conversion of function application shows how to use the natural numbering of the nodes of the typing element for referring to the attributes of different nodes:

HaskellAttrib for $\boxed{\textbf{Standard.@}}$

```
    expr(1) = expr(2) ++ ' ' : pexpr(3)
    pexpr(1) = '(' : expr(1) ++ ")"
```

In the definition for λ-abstraction, we have to take care whether there is a bound variable or not; we choose to use different conversions for this purpose. This is implemented via the built-in macro *bvar* which takes five arguments: The first argument refers to a node; if this node has a bound variable, then the call evaluates to the fourth argument in an environment where the second argument, considered as a macro name, is bound to the result of applying the third argument to the bound variable. Otherwise it evaluates to the fifth argument. E.g., if `@1` refers to a binder having a bound variable node with the built-in *number* attribute being `1005`, then the macro call "*bvar*(*1*,`bv`,*number*,`"x`*bv*`"`,`[]`)" evaluates to "`"x1005"`". If `@1` however refers to a binder that does not bind any variable (as e.g. in $\lambda\, x.\, 3$), then that macro call evaluates to "`[]`".

Here this is used to implement the distinction between a λ-abstraction in Haskell and an application of `const` — since λ-abstractions already need parentheses in our context, we distinguish globally for *expr* and *pexpr* together:

HaskellAttrib for $\boxed{\lambda}$

```
bvar(1,bv,expr,
expr(1) = "(\\ " ++ bv ++ " -> " ++ expr(2) ++ ")"
pexpr(1) = expr(1)
,
expr(1) = const pexpr(2)
pexpr(1) = '(' : expr(1) ++ ")"
)
```

Since HOPS and Haskell are quite different in several respects, the full conversion mechanism has to take care that

- it introduces distinct variable names for distinct variables in the same scope,
- it converts sharing inside a rule's right-hand side into `where`-clauses or, if the shared subexpression contains a λ-bound variable, into `let`-bindings,
- it converts sharing between the two rule sides into as-patterns if appropriate,
- it enforces the capitalisation conventions of Haskell, since in HOPS there are no restrictions on the syntax of node labels.

The following Haskell code was automatically generated (with manually added line breaks) from the declarations and rules shown in the module editor picture in the previous section:

```
mapT :: (Vertex -> a1 -> a3) -> (Table a1) -> Table a3
mapT v9 v11 = qTable_a (array (bounds v23)
  (map (\ v29 -> (v29, v9 v29 (v23 ! v29))) (indices v23)))
 where v23 = qTable_r v11

vertices :: Graph -> [Vertex]
vertices v4 = indices (qTable_r (qGraph_r v4))

edges :: Graph -> [(Vertex, Vertex)]
edges v7 = concat (map (\ v15 ->
    map (qMkpa1 v15) (qTable_r (qGraph_r v7) ! v15))
                       (vertices v7))
```

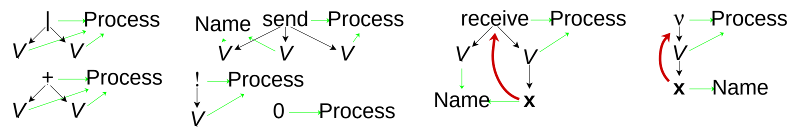
It should be noted that the details of this conversion of HOPS to Haskell are not hard-coded into HOPS itself, but employ the attribution mechanism which allows to program such conversions. Therefore conversions of different HOPS languages (as defined in different sets of HOPS modules) into different target languages can be defined by any user of HOPS; we shall see an example for this in the next section.

5 A Language Experiment: The π-Calculus in HOPS

In an early prototype of HOPS, P. Kempf demonstrated how to translate the π-calculus into a HOPS language and how to translate a part of that language (omitting choice) into Ada tasks, see the chapters 7 and 8 of the HOPS report by Bayer et al. (1996). In the current version, the definition of the π-calculus as a HOPS language is now much more direct and natural, and also the conversion to Ada profits from the powerful attribution mechanism implemented presented in Sect. 4.

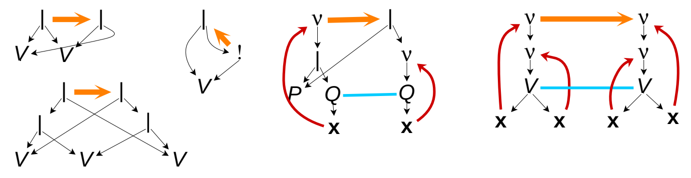
The π-calculus emerged as an advanced form of a process-calculus describing features of concurrent computation. It formalises synchronous handshake communication over links, where, in contrast to its predecessor CCS, the communicated values are links themselves. A lot of work, theoretical as well as practical, has been influenced by the π-calculus. It has strong connections to formal theories like linear logic (Bellin and Scott 1994), but moreover has been shown to be a useful tool in the design of concrete distributed systems (Orava and Parrow 1992). Our formulation is oriented at the algebraic form of the π-calculus as presented by Milner (1993).

The language of the π-calculus only knows two types: Process for processes and Name for link or channel names. Processes can be built using the following constructors:



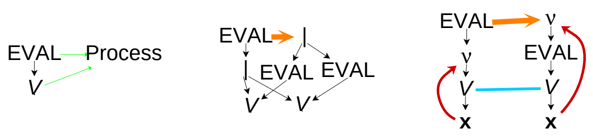
0 is the process of *inaction*; | is concurrent composition; + is choice, and ! is replication. receive is usually called “positive prefixing”and written $x(y).P$ meaning reception along channel x of a symbol which is bound to y in the subsequent process P. send is “negative prefixing” written $\bar{x}y.P$, meaning the export of the link y along channel x. The “restriction” ν serves to limit the visibility of channels; ν and receive both can bind variables.

There are many algebraic equalities that hold in the π-calculus and which have to be translated into HOPS rules; we show only a few:



However, reduction via *communication* in the π-calculus does not give rise to an equality, and may therefore not be applied at arbitrary places in a process term.

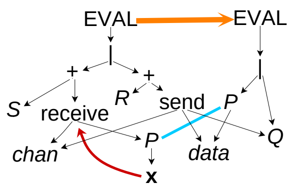
Since HOPS currently does not provide any other means to implement such a restriction to rule application, we use a trick to encode the places where communication is legal: The complete process is crowned with an EVAL tag, and the closure of the reduction wrt. concurrent composition and restriction is reflected by rules that propagate that tag:



The communication rule itself, usually given as

$$(R + c(x).P) \mid (S + \overline{c}d.Q) \rightarrow P\{d/x\} \mid Q,$$

then requires and preserves the presence of this evaluation tag:



For the conversion to Ada, processes are implemented as tasks, and links are task references — we leave out the details of these types and only show the definitions for positive and negative prefixing (send and receive). The *proc* attribute has values of type `Int -> (String,Int)` where the integers are auxiliary inherited and synthesised attributes of the *unfolded tree*, not of the potentially shared term graph, and the `String` is an Ada program fragment.

Sending is implemented by a call to the `send` entry of the task referred to by the channel argument, transferring the data argument as argument to that entry. This is composed sequentially with the remaining process:

HaskellAttrib for $\boxed{\textbf{send}}$

```
proc(1) n =
  (name(2) ++ ".send(" ++ name(3) ++ ";\n" ++ p4a
  ,p4b)
 where (p4a,p4b) = proc(4) n
```

For receiving, we create a new block with a local variable `a` for the received link name, and call the `receive` entry of the channel argument, submitting `a` as a result parameter:

HaskellAttrib for $\boxed{\textbf{receive}}$

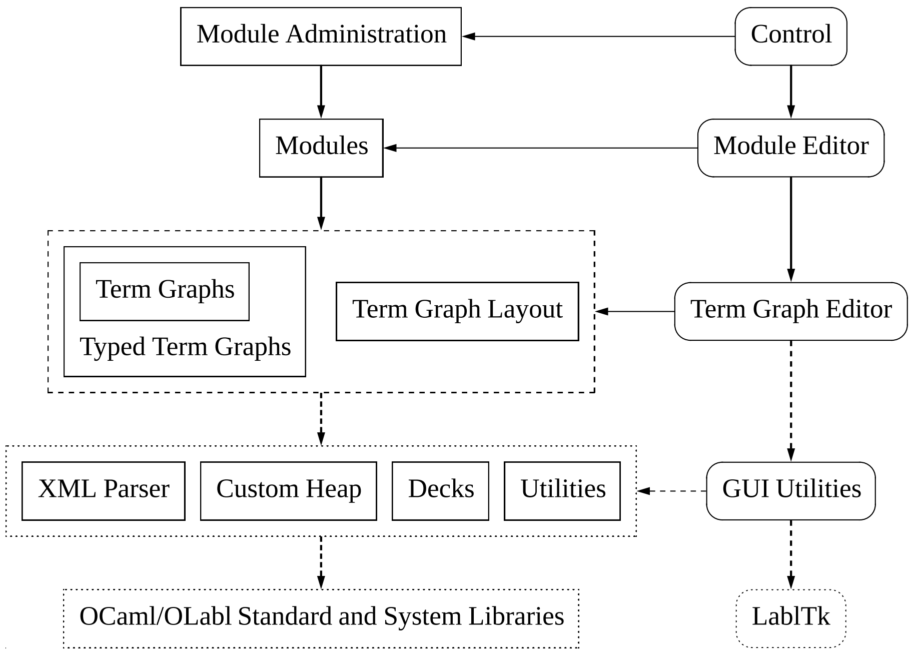
```
proc(1) n =
 ("B" ++ show n ++ ":declare\n" ++
  "    a:Name;\n" ++
  "   begin\n" ++
  "    " ++ chan ++ ".receive(" ++ chan ++ ", " ++ bname ++ ");\n"
  "    " ++ p3a ++ "\n" ++
  "   end B" ++ show n ++ ";\n"
 ,p3b)
 where (p3a,p3b) = proc(3) (n+1)
       chan = name(2)
       bname = bvar(1,bv,name,bv,"xlnumber")
```

6 Implementation

Since HOPS is an interactive system, response is essential. This also had to be taken into account for the choice of the implementation language. Work on the current version of HOPS was started in CSL (Caml Special Light), a re-implementation of Caml-Light with improved performance and a powerful module system. CSL later added object-oriented features (which are not yet taken advantage of in the implementation of HOPS) and changed name to OCaml; for the graphical interface HOPS relies on the type-safe interface to Tk provided by the labelised variant OLabl.

The choice of OCaml/OLabl for programming and of LablTk for the user interface has fulfilled all the expectations with respect foremost to ease and elegance of programming, but also access to external tools and even performance — on the whole, the speed is very acceptable for an interactive programming system: the only relatively slow operation is the calculation of a new layout; switching off intermediate drawing during automatic transformation more than doubles the speed (to about 25 transformations per second on simple examples and on a Sun Ultra).

The following picture gives a rough overview of the way the implementation of HOPS is structured:



The lowest level is provided by the implementation basis, i.e., by the OCaml/OLabl distributions, and has mainly been included in the drawing for differentiating more clearly between the "logical" kernel of HOPS (drawn to the left), which does not require access to the Tk interface LablTk, and the graphical user interface components (drawn to the right), which do.

Therefore, the kernel can also be used in stand-alone applications that can be linked without the Tk libraries. Currently there are three kinds of functionality of HOPS made available via stand-alone tools in this category:

– module document output generation for printing modules as documents, with optional call of the back-end (Lout or LaTeX/dvips) for PostScript generation,
– module dependency graph generation, and
– attribute definition generation for whole modules, with optional attribute evaluation by an external interpreter (`hugs`).

The *module administration* component is responsible for loading modules on demand and keeping information about modules. Since in the full system this information includes possible active module editors and therefore GUI components, the OLabl module implementing module administration essentially provides a functor that accepts this additional information as its argument — in the stand-alone utilities mentioned above a trivial argument is supplied.

The *module* component provides the internal data structures and utilities for the representation of HOPS modules, that is of documents with embedded *entries* such as declarations, rules, cross references, and attribution definitions.

The inner kernel of HOPS manages the *term graphs* contained in declarations, rules, and example graphs. An inner level allows construction, access and manipulation of all term graph components including binding, variable identity and typing. This is wrapped up in an outer layer that is responsible for the consistency of binding and variable identity and most notably for well-typedness. This outer layer also contains all advanced construction functions and the rule matching and application mechanisms.

Graph layout is stored in a separate data structure that is kept alongside its graph in the module structure. Layout is currently calculated externally by `dot` (Gansner et al. 1993) from AT&T's graphviz tool suite, but a more specialised layout mechanism specifically designed for the needs of HOPS term graphs is being developed in HOPS.

The nodes of HOPS term graphs need identifiers that can be used as keys in finite mappings for many purposes. Since references in OCaml are neither ordered nor hashable, it is not feasible to use just references for allocation of HOPS nodes, so we had to introduce our own interface to a heap management that uses appropriate keys.

Besides many simple extensions of the standard library, we also use a data structure of mixed decks that are employed for managing the text flow with embedded entries that constitutes the direct contents of HOPS modules. This data structure is wrapped into a functor that takes the monoid of the text flow as its argument and returns a double-ended doubly-linked alternating queue data type that is parametric in the type of the embedded items and which makes use of the monoid composition e.g. when deleting entries.

The data structures of the kernel of HOPS are made accessible through the corresponding parts of the graphical user interface.

The *term graph editor* is essentially a Tk canvas associated with a term graph and a layout for that term graph and with other internal state, equipped with event bindings and pop-up menus that make operations on all these components accessible.

The *module editor* is, as shown in the picture in Sect. 7, a window containing a text widget in which the module document is displayed. Entries are displayed as embedded widgets, and the display of entries that contain term graphs contains a term graph editor.

The user accessible functions of the module administration, such as deliberate loading of modules or generation of new modules, together with a set of configuration functions are accessible from a separate *control window*, which is the only window displayed when starting HOPS.

7 Concluding Remarks

There is still much ongoing work on HOPS. Since HOPS is mainly intended as a program development environment where eventually usable code in more or less arbitrary languages can be generated, the actual possibility of Haskell conversion can only be considered as a first step into that direction. Besides the attribution mechanism already implemented, the key ingredient of successful code generation will be powerful transformation programming capabilities.

We are also considering more powerful type systems, including Haskell type classes, unique types, polytypic programming and explicit polymorphism (Kahl 1998).

More information, including the HOPS distribution and manuals, is available from the HOPS home page at URL:

http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/

References

Bayer, A., Grobauer, B., Kahl, W., Kempf, P., Schmalhofer, F., Schmidt, G., Winter, M. (1996): The **H**igher-**O**bject **P**rogramming **S**ystem "**HOPS**". Internal Report, Fakultät für Informatik, Universität der Bundeswehr München. URL http://inf2-www.informatik.unibw-muenchen.de/HOPS/papers/HOPSreport-1996.html. 214 pp.

Bellin, G., Scott, P.J. (1994): On the π-calculus and linear logic. *Theoretical Computer Science* **135** (1), 11-63.

Gansner, E.R., Koutsofios, E., North, S.C., Vo, K. (1993): A Technique for Drawing Directed Graphs. *IEEE-TSE*.

Kahl, W. (1994): Can Functional Programming Be Liberated from the Applicative Style?. In Bjørn Pehrson, Imre Simon (eds.), *Technology and Foundations, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August – 2 September 1994, Volume I*, pages 330–335. IFIP Transactions. North-Holland. IFIP.

Kahl, W. (1996): *Algebraische Termgraphersetzung mit gebundenen Variablen*. Reihe Informatik. Herbert Utz Verlag, München. ISBN 3-931327-60-4, zugleich Dissertation an der Fakultät für Informatik, Universität der Bundeswehr München.

Kahl, W. (1998): The **H**igher **O**bject **P**rogramming **S**ystem — User Manual for HOPS, Fakultät für Informatik, Universität der Bundeswehr München. URL http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/.

Kahl, W. (1998): Internally Typed Second-Order Term Graphs. In Juraj Hromkovic, Ondrej Sýkora (eds.), *Graph Theoretic Concepts in Computer Science, 24th International Workshop, WG '98, Smolenice Castle, Slovak Republic, June 1998, Proceedings*, pages 149–163. LNCS 1517. Springer-Verlag.

Kingston, J.H. (1998): *A User’s Guide to the Lout Document Formatting System (Version 3.12).* Basser Department of Computer Science, University of Sydney. ISBN 0 86758 951 5. URL ftp://ftp.cs.su.oz.au/jeff/lout.

Klop, J.W. (1980): Combinatory Reduction Systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam. PhD Thesis

Klop, J.W., van Oostrom, V., van Raamsdonk, F. (1993): Combinatory reduction systems: introduction and survey. *Theoretical Computer Science* **121** (1–2), 279–308.

Knuth, D.E. (1984): Literate Programming. *The Computer Journal* **27** (2), 97–111.

Milner, R. (1993): The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer, H. Schwichtenberg (eds.), *Logic, Algebra, and Specification*, pages 203–246. Vol. 94 of NATO Series F: Computer and Systems Sciences. Springer.

Orava, F., Parrow, J. (1992): An Algebraic Verification of a Mobile Telephone Network. *Formal Aspects of Computer Science* **4**, 497–543.

Williams, R.N. (1992): FunnelWeb User’s Manual. URL http://www.ross.net/funnelweb/introduction.html. Part of the FunnelWeb distribution

Zierer, H., Schmidt, G., Berghammer, R. (1986): An Interactive Graphical Manipulation System for Higher Objects Based on Relational Algebra. In *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 68–81. LNCS 246. Springer-Verlag, Bernried, Starnberger See.