Graphs for Recording Type Information

Bruce J. McAdam

Technical Report ECS–LFCS–99–415
Laboratory for Foundations of Computer Science
Division of Informatics
The University of Edinburgh

November 15, 1999

Abstract

This report presents a way of recording information about the types of programs as a graph. The focus is on programs typed with Hindley-Milner type systems. The graphs can record information about untypeable programs (which are traditionally rejected with a type error message by type inference algorithms) and thus can be used to produce information to help programmers debug programs.

1 Introduction

This report is about a way of recording information about the types of programs. Programmers frequently want to know about the types in their programs so that they can repair mistakes and understand their code better. A number of authors have looked at particular ways of presenting type information to programmers. This work is summarised in Section 2. The method presented in this paper records information about typeable or untypeable programs as graphs. Significantly, we treat typeable and untypeable programs in the same way. All other work has dealt either with typeable or untypeable programs. Examples of graphs are given in Section 3. The algorithms for generating graphs and extracting information from them are given in Section 4. An accompanying paper [McA99] shows how the information presented by other authors can be extracted from graphs. All the work described has been implemented in Standard ML for a λ-with-let calculus. Appendix A contains source code for the implementation. The conclusions of this report are summarised in Section 5.

2 Related Work

One of the earliest papers related to the topic of acquiring information about types is by Johnson and Walz [JW86]. They consider the general issue of unification failure. This is relevant to type errors as most type inference algorithms detect the presence of a type error by failure to unify two types. Johnson and Walz treat types as graphs and use flow analysis to find the best fit for unification. Implemented in a development environment, this suggests what the type of an untypeable part of the program is likely to be, for example if an identifier was used several times with type int, and once with type string it is likely that the correct type is int. The programmer can then use this information to make changes to the program. The use of graphs in Johnson and Walz's work is extremely suggestive of the approach in this paper, but no attempt has yet been made to consolidate them.

Wand [Wan86] presented a scheme for locating the probable mistake in a program. This is in contrast to the location given by compiler error messages which is the point the compiler was examining when it found that there was a type error. This is not necessarily close to where the programmer has made a mistake. The connection Wand's technique and this report is examined in more detail in [McA99].

Smaller pieces of work in this area include Turner's [Tur90] compiler with improved error messages. This printed out more types, and with better accompanying messages when the conventional inference algorithm failed because of a unification failure. Soosaipillai [Soo90] produced a type checker which could be interrogated about the origins of derived types. A programmer using the system could ask which parts of the program were used to derive the type of another part. This is an example of a type explanation system.

Beaven and Stansifer [BS93] provide a simple form of error explanation. When two types fail to unify during type inference, the system gives an explanation of how each of the two types was derived.

Duggan and Bent [DB96] explain type inference by recording changes made to types during unification. Duggan [Dug98] has further extended this work by defining the notion of a correct type explanation. In [McA99] I discuss how correct type explanations may be produced from the scheme presented in this paper.

Bernstein and Stark [BS95] take a distinctive approach, inspired by Shao and Appel [SA93]. They propose not only a new inference algorithm, but new static semantics. Their method centres on *assumption environments*: mappings of identifiers to sets of types. An assumption environment characterises the uses of free variables in a program fragment. The inference algorithm takes open expressions and returns the type of every instance of the free identifiers. This system is further examined in [McA99], and I show that this paper's technique can simulate Bernstein and Stark's.

Lee and Yi [LY98] examine a top down type inference algorithm (the conventional algorithm is bottom up) and show that it detects errors after examining less of the program than bottom up algorithms. It has been stated by Yi [Yi99] and Leroy [Ler99] and Duggan [Dug98] that top down and bottom up inference algorithms each give better error messages in different situations. In [McA98] I presented a way of removing the left-to-right bias from both top-down and bottom-up type inference algorithms, the graph generation algorithm in this report follows that technique.

3 Graphs

The structure of graphs follows the structure of types (they are not annotated syntax trees). There are vertices representing the types of expressions and type constructors. The varieties of vertex are given in Definition 1 and illustrated in Figure 1.

Figure 1 Sample vertices. A vertex for the program fragment $\lambda i.i$; the fragment $\lambda i.i$ tagged with an instance of I; a nullary type-constructor, int; a unary type-constructor, list; the binary function type-constructor; and a type variable vertex, α.

$\circ\ \lambda i.i$ $\quad$ $\circ\ [\lambda i.i]I$ $\quad$ int $\quad$ $\circ\ \mathsf{list}$ $\quad$ $\circ \rightarrow \circ$

Definition 1 (Varieties of vertex)

$$\begin{array}{rcll}
v & ::= & f & \textit{A program fragment} \\
 & | & [f_0]_{f_1, f_2 \ldots f_n} & \textit{A program fragment tagged by other program fragments} \\
 & | & (\circ_1 \ldots \circ_n)c_i & \textit{A type constructor } c \textit{ with arity } n \textit{ and unique identifier } i \\
 & | & c_i.j & \textit{The } j\textit{th connection point of the vertex } (\circ_1 \ldots \circ_n)c_i \\
 & | & \alpha & \textit{A type variable}
\end{array}$$

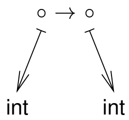
Program fragment vertices. These are identified by a program fragment, f. A fragment is a subexpression of the program and its location within the program, *i.e.* a node of the syntax tree. These may also be tagged by a sequence of other fragments to give $[f_0]_{f_1, f_2 \ldots f_n}$. The tagging is used to implement Hindley-Milner polymorphism. These vertices represent the types of fragments.

Type constructor vertices. These are identified by a type constructor and some unique identifier. For example, there may be several function type constructor vertices in a graph each with its own identifier. This variety of vertex contains a number of sub-vertices called *connection points*. Function type constructor vertices are denoted $\circ \rightarrow_i \circ$ with two connection points, and an identifier i. The connection points may be referred to as $\rightarrow_i .0$ and $\rightarrow_i .1$. In general a type constructor vertex c with identifier i and arity n is $(\circ_1 \ldots \circ_n)c_i$. There is a special nullary type constructor $\mathsf{unbound}$.

Type variable vertices. These are identified by some unique identifier and written α (much like type variables in traditional type inference).

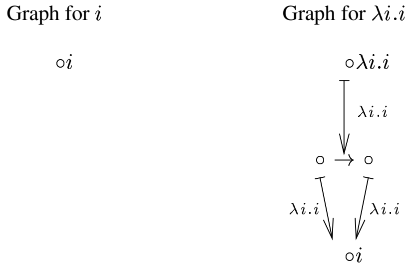
Types are built up by edges coming from connection points, as in Figure 2.

Figure 2 The type $\mathsf{int} \rightarrow \mathsf{int}$



Edges are added between vertices which must represent the same type. Written $\overset{f}{\mapsto}$, they are labelled by program fragments and cannot go from type constructor or type variable vertices. They can, however, go from the connection points of type constructor vertices. Figure 3 shows how the graph for the identity function is generated. First the graph for the subexpression i is generated (this is a single vertex). Then the $\lambda i.i$ and $\circ \rightarrow \circ$ vertices are added. The type of $\lambda i.i$ must be a function type, so an edge is added from $\lambda i.i$ to $\circ \rightarrow \circ$ (labelled with the expression of current interest — $\lambda i.i$). The first connection point is connected to the vertex representing the type of the argument — *i.e.* the i vertex, and the second connection point is connected to the vertex for the result expression — also i.

Figure 3 Generating a graph for the identity function. Read as $\alpha_i \rightarrow \alpha_i$.

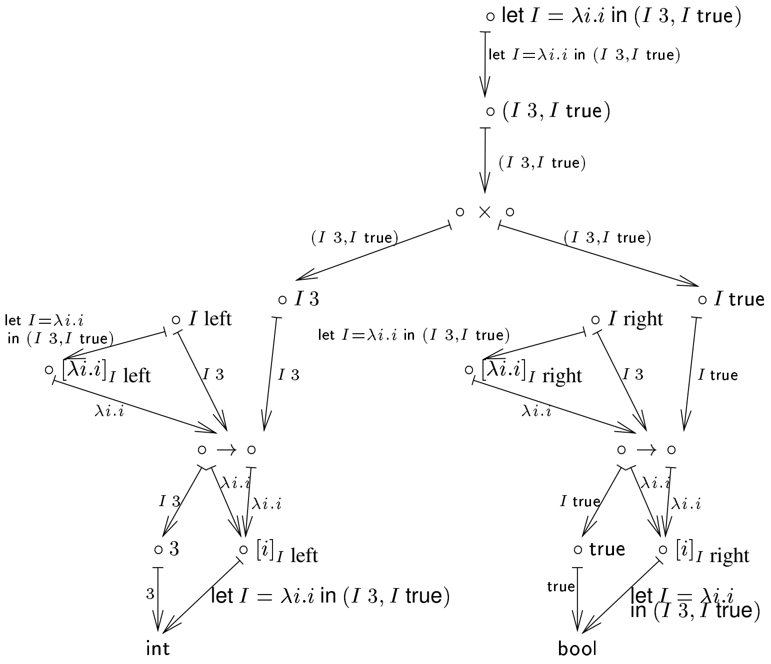


A more complex example is given in Figure 4. This is a let expression which contains an application. There are several features to observe.

- The graph for $\lambda i.i$ (see Figure 2) appears twice, labelled with each occurrence of I.

- The application expressions tell us that each occurrence of I must be a function.
- The application expressions tell us how the types of I should be instantiated.
- Tuples are represented in the obvious way.

Figure 4 A let expression. The type is read as int $\times$ bool.



Graphs can be generated for untypeable programs. The program in Figure 5 is similar to that of the previous figure, except that I is λ bound instead of let-bound — this makes the program untypeable. From the graph, we can see roughly what the type should be (a function returning a tuple).

The difference between the patterns of edges in Figures 4 and 5 are shown in Figure 6. Sometimes when a vertex has several edges from it, they all ultimately meet up, other times they diverge. When edges diverge, we call it a branch. Branches indicate that programs are untypeable. There is a conflict between two types, corresponding to a unification failure in conventional type inference.

A program may also be untypeable if it has unbound identifiers. Figure 7 shows the graph for a program with unbound identifiers. The unbound identifiers are marked with unbound. In [McA99] I explain how to read the required types of unbound identifiers following the example of [BS95].

The final way in which a graph can indicate that its program is untypeable is if it contains a cycle, as in Figure 8. This corresponds to an occurs error in traditional type inference. Cycles are indirect, as they can span across the gaps between type constructor vertices and their connection

Figure 5 An untypeable program.

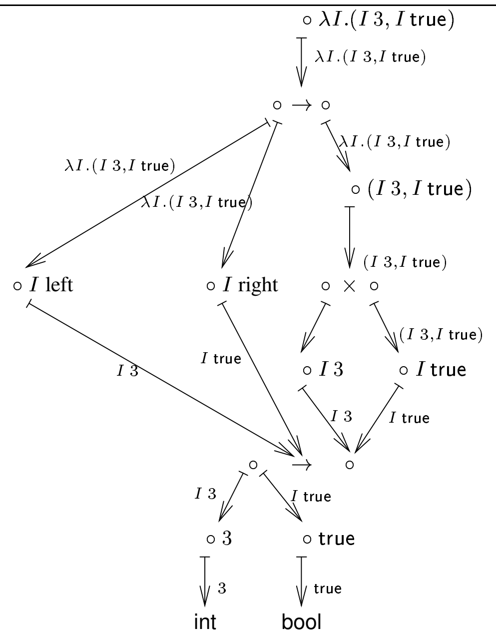
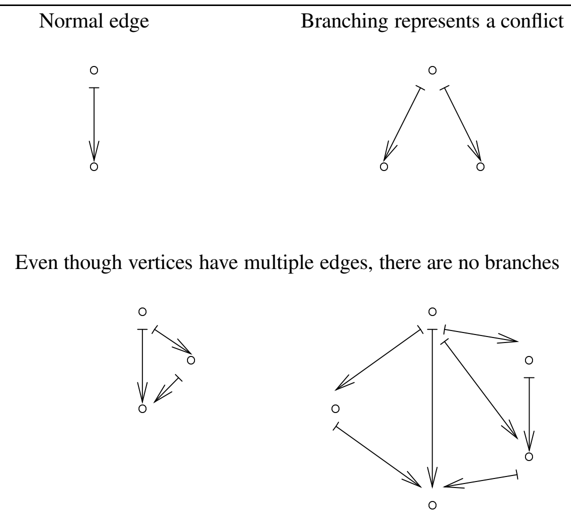


Figure 6 Patterns of edges



points, *e.g.* in the figure, it is not possible to reach any vertex from itself, but it is possible to reach the lower arrow from its own connection point.

Figure 7 An open expression

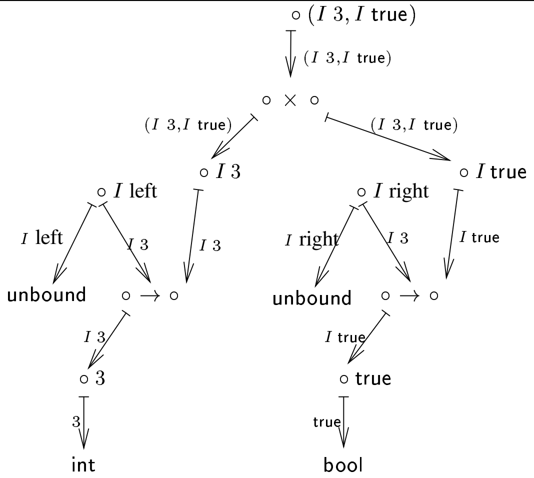
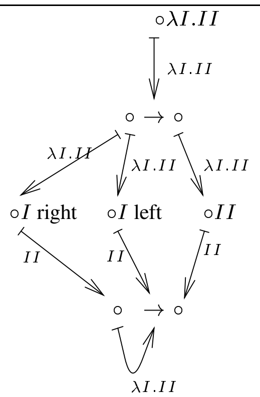


Figure 8 A graph with a cycle



4 Algorithms

Analysis of programs using graphs is a two stage process. First generate a graph, then read useful information from the graph. First we will look at the algorithm for generating graphs. In Section 4.2 we will see how to read typings from graphs, the paper [McA99] shows how to read other information from graphs.

4.1 Generating Graphs

There are several important algorithms for generating graphs: fine scale traversal, closing the graph, and the actual generation algorithm. Once a graph has been generated, the algorithms in Section 4.2 tell us how to interpret it.

4.1.1 Fine Scale Traversal

Generating and reading graphs requires an operation which searches for all the type constructor vertices, type variable vertices and other vertices without children, reachable from some root vertex. The algorithm for this can be found in Figure 9.

Figure 9 Algorithm for finding important vertices starting from a root. Takes a graph and vertex, returns three sets of vertices.

$$
\begin{array}{rcl}
\mathsf{search}(G, (\circ_1, \cdots, \circ_n)c) & = & (\{(\circ_1, \cdots, \circ_n)c\}, \{\}, \{\}) \\
\mathsf{search}(G, \alpha) & = & (\{\}, \{\alpha\}, \{\}) \\
\mathsf{search}(G, v) & = & \textbf{if } \mathsf{children}(G, v) = \{\} \textbf{ then} \\
& & \quad (\{\}, \{v\}) \\
& & \textbf{else} \\
& & \quad \textbf{let} \\
& & \quad\quad r = \bigcup\{\mathsf{search}(G, v') : v' \in \mathsf{children}(G, v)\} \\
& & \quad \textbf{in} \\
& & \quad\quad (\bigcup\{cs : (cs, tvs, vs) \in r\}, \\
& & \quad\quad \bigcup\{tvs : (cs, tvs, vs) \in r\}, \\
& & \quad\quad \bigcup\{vs : (cs, tvs, vs) \in r\})
\end{array}
$$

The three sets returned by search are:

Type constructor vertices. If this contains several instances of any type constructor then these instances should be merged to close the graph.

Type variable vertices. If there are several type variables found then they can be merged. If any type constructors were found then the type variables can be removed.

Other vertices with no edges coming from them. These should have edges attached to the type constructor or type variable vertices.

If search returns more than one vertex, then the graph currently contains a branch at the given root vertex.

The search function is used whenever the graph must be traversed. We use it to ignore vertices which have edges from them and therefore have been instantiated as another type. In most type inference algorithms based on references, chains of references are eliminated (aliasing), but in the form of inference in this paper we keep the vertices corresponding to these chains of references as they represent valuable information.

The search function stops at type constructors. The remaining functions will traverse the graph through type constructors to their connection points.

4.1.2 Closing a graph

The closure operation is illustrated on an example in Figure 10. It is used to ensure that:

- There is at most one instance of any type constructor reachable from a vertex
- If a type constructor and some other vertex are reachable from any vertex then there is a path from the other vertex to the type constructor.

In order to do this, closure merges distinct instances of type constructors, and adds edges from other vertices to type constructors.

Figure 10 Adding an edge to a graphs

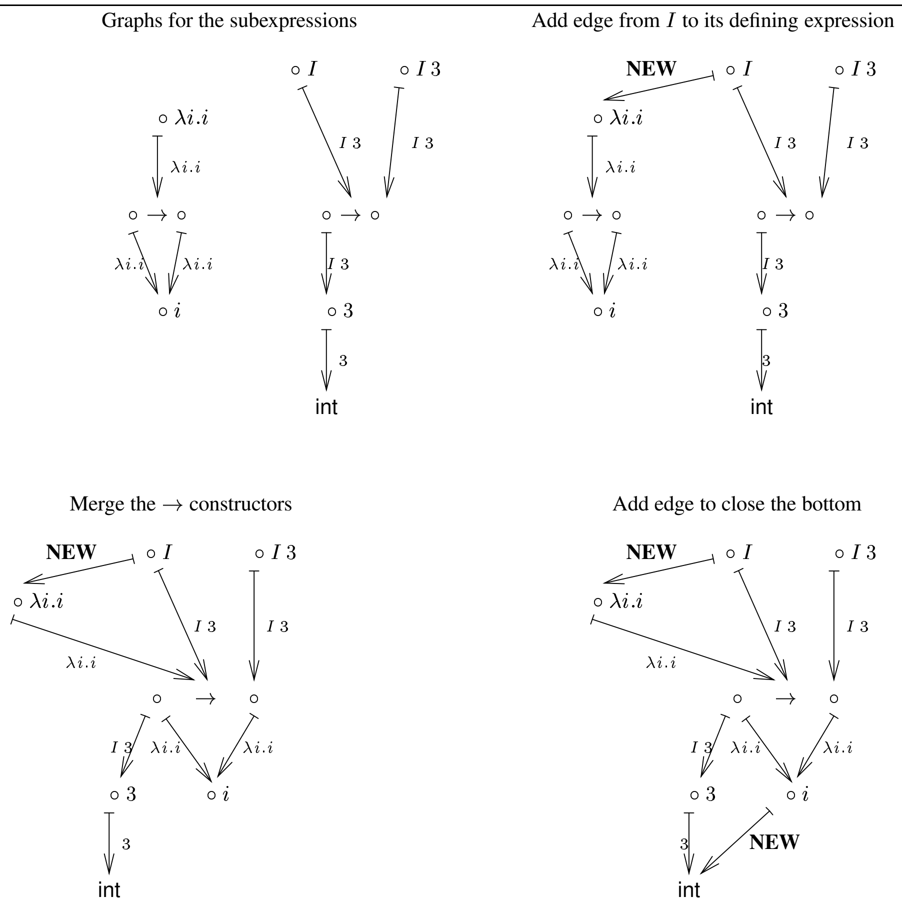


Figure 11 shows the closure algorithm, and Figure 12 the accompanying merging algorithm (the two are mutually recursive).

There are four possible results to search in the main case of close. If there are no other vertices reachable, then do not do anything to the graph (just return an updated list of vertices seen). If there are several undistinguished vertices reachable, then create a new type variable vertex and link to it (*e.g.* if $x \mapsto y$ and $x \mapsto z$ then we must have $y \mapsto \alpha$ and $z \mapsto \alpha$ to eliminate the

Figure 11 The closure algorithm. Takes a graph, vertex, label for new edges and set of vertices seen so far (initially this should be empty). Returns updated graph, and updated list of vertices seen.

$$
\begin{array}{rcl}
\mathsf{close}((V,E),(\circ_1\cdots\circ_n)c,l,s) & = & \textbf{if } (\circ_1\cdots\circ_n)c \in s \textbf{ then } ((V,E),s) \\
 & & \textbf{else } \mathsf{closeSet}((V,E),\{\circ_1\cdots\circ_n\},l) \\
\mathsf{close}((V,E),v,l,s) & = & \textbf{if } v \in s \textbf{ then } ((V,E),s) \textbf{ else} \\
 & & \textbf{case } \mathsf{search}(v,(V,E) \\
 & & (\{\},\{\},\{v'\}) \Rightarrow (G, s \cup \{v'\}) \\
 & & (\{\},\{\},vs) \Rightarrow \textbf{let} \\
 & & \qquad\qquad \alpha = \mathsf{newTyvarVertex}() \\
 & & \qquad \textbf{in} \\
 & & \qquad\qquad ((V \cup \{\alpha\}, E \cup \{(v,l,\alpha) : v \in vs\}), s \cup \{v\}) \\
 & & (\{\},\{\alpha\} \cup tvs, vs) \Rightarrow \textbf{let} \\
 & & \qquad\qquad E_0 = \{(v_0,l,v_1) : (v_0,l,v_1) \in E \wedge v_1 \notin tvs\} \\
 & & \qquad\qquad E_1 = \{(v_0,l,\alpha) : (v_0,l,\beta) \in E \wedge \beta \in tvs\} \\
 & & \qquad\qquad E_2 = \{(v_0,l,\alpha) : v_0 \in vs\} \\
 & & \qquad \textbf{in} \\
 & & \qquad\qquad ((V - tvs, E_0 \cup E_1 \cup E_2), s \cup \{v\}) \\
 & & (cs, tvs, vs) \Rightarrow \textbf{let} \\
 & & \qquad\qquad (cs', ((V',E'),s')) = \mathsf{merger}((((V,E),s),cs,l) \\
 & & \qquad\qquad E'' = \{(v_0,l,c) : (v_0,l',\alpha) \in E' \wedge \alpha \in tvs \wedge c \in cs'\} \\
 & & \qquad\qquad E''' = \{(v_0,l,c) : v_0 \in vs \wedge c \in cs'\} \\
 & & \qquad \textbf{in} \\
 & & \qquad\qquad ((V' - tvs, E'' \cup E'''), s')
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{closeSet}((G,s),\{\},l) & = & (G,s) \\
\mathsf{closeSet}((G,s),\{v\} \cup V,l) & = & \mathsf{closeSet}(\mathsf{close}((G,s),v,l),V,l)
\end{array}
$$

branch). If there are several type variables (and other vertices), then remove all but one type variable and put all edges to all the type variables to the nominated type variable (and connect all other vertices to the remaining type variable). If there are type constructors reachable, then merge all similar type constructors (*i.e.* merge all $\circ \to \circ$ vertices and all int vertices), remove the type variables connecting their edges to all the type constructors (type variables are removed if the actual type is known), and connect any other vertices to all the type constructors.

Figure 12 Merger. Takes a graph and a set of type constructor vertices, merges all similar type constructors in the graph.

$$
\begin{array}{rcl}
\mathsf{merger}((G,s),\{\},l) & = & ((G,s),\{\}) \\
\mathsf{merger}(G,\{(\circ_1\cdots\circ_n)c\}\cup cs,l) & = & \textbf{let} \\
& & \quad like = \{c' : c' \in cs' \wedge \mathsf{tycon}(c') = \mathsf{tycon}(c) \\
& & \quad unlike = \{c' : c' \in cs' \wedge \mathsf{tycon}(c') \neq \mathsf{tycon}(c) \\
& & \quad E_0 = \{(v_0,l,c) : (v_0,l,c') \in E \wedge c' \in like\} \\
& & \quad E_1 = \{(v_0,l,c.\iota) : (v_0,l,c'.\iota) \in E \wedge c' \in like\} \\
& & \quad E_2 = \{(v_0,l,v_1) : (v_0,l,v_1) \in E \wedge v_1 \notin like \wedge \nexists c \in like.c : \iota = v_1\} \\
& & \quad E' = E_0 \cup E_1 \cup E_2 \\
& & \quad E'_0 = \{(c.\iota,l,v_1) : (c'.\iota,l,v_1) \in E' \wedge c' \in like\} \\
& & \quad E'_1 = \{(v_0,l,v_1) : (v_0,l,v_1) \in E' \wedge \nexists c \in like : c.\iota = v_0\} \\
& & \quad E'' = E'_0 \cup E'_1 \\
& & \quad (G',s') = \mathsf{close}(((V - like, E''),s),c) \\
& & \quad (cs'',(G'',s'')) = \mathsf{merger}((G',s'),unlike) \\
& & \textbf{in} \\
& & \quad (cs'' \cup \{c\},(G'',s''))
\end{array}
$$

merger takes a set of vertices. It picks a vertex and finds all the vertexes like it in the set, then removes all the similar vertices from the graph and set and connects their edges to the remaining one then closes below the remaining one. It repeats this until no vertices are left in its set.

4.1.3 Generating a graph

The graph generation algorithm in Figure 13 is quite simple. By closing whenever an edge is added, there is no need for an explicit call to a unification function and because substitutions and types are combined in one data structure there is no need for explicit operations on substitutions.

The algorithm makes use of a function, $\mathsf{free}(e,x)$, which returns every syntactic instance of x in e. These subexpressions will be vertices in the graph for e.

generate must also make use of a type environment which keeps track of which identifiers are in scope and which refer to the basis environment. Γ is a pair, (I,B), of a set of bound identifiers, I, and a basis, B, mapping identifiers to a graph and vertex pair, (G,v). If generate encounters an identifier in I then it will produce a one-node graph, if the identifier is in B then the corresponding graph in B is used. If an identifier is in neither I nor B a graph representing the unbound type is returned. In generate, identifiers are added to I but B is never modified.

The last case of generate (for let expressions) is the most complex as it must deal with polymorphism. The graphs for the definition and use subexpressions are generated. A vertex for the let expression is added to the graph, and an edge connects it to the use expression vertex. The

Figure 13 Generating a graph

$$
\begin{array}{rcl}
\mathsf{generate}((I,B),x) & = & \textbf{if } x \in I \textbf{ then } ((\{x\},\{\}),x) \\
& & \textbf{else, if } (x,((V,E),v)) \in B \textbf{ then } ((V \cup \{x\}, E \cup \{x,x,v\}),x) \\
& & \textbf{else (}x\textbf{ is unbound) let } v = \mathsf{vertex}(\mathsf{unbound}) \textbf{ in } (\{x,v\},\{(x,x,v)\}) \\
\mathsf{generate}((I,B),\lambda x.e_0) & = & \textbf{let} \\
& & \quad (V_0,E_0) = \mathsf{generate}((I \cup \{x\},B),e_0) \\
& & \quad v = \mathsf{vertex}(\rightarrow) \\
& & \quad V = V_0 \cup \{\lambda x.e_0, v\} \\
& & \quad E = E_0 \cup \{(\lambda x.e_0, \lambda x.e_0, v),(v.2,\lambda x.e_0,e_0)\} \cup \\
& & \qquad \{(v.1,\lambda x.e_0,e) : e \in \mathsf{free}(e_0,x)\} \\
& & \textbf{in} \\
& & \quad \mathsf{close}((V,E),\lambda x.e_0,\lambda x.e_0) \\
\mathsf{generate}(\Gamma,e_0e_1) & = & \textbf{let} \\
& & \quad (V_0,E_0) = \mathsf{generate}(\Gamma,e_0) \quad (V_1,E_1) = \mathsf{generate}(\Gamma,e_1) \\
& & \quad v = \mathsf{vertex}(\rightarrow) \\
& & \quad V' = V_0 \cup V_1 \cup \{v, e_0e_1\} \\
& & \quad E' = E_0 \cup E_1 \cup \{(e_0e_1,e_0e_1,v.2),(v.1,e_0e_1,e_1),(e_0,e_0e_1,v)\} \\
& & \textbf{in} \\
& & \quad \mathsf{close}((V',E'),e_0,e_0e_1) \\
\mathsf{generate}((I,B),\mathsf{let} x = e_0 \mathsf{in} e_1) & = & \textbf{let} \\
& & \quad (V_0,E_0) = \mathsf{generate}((I,B),e_0) \quad (V_1,E_1) = \mathsf{generate}((I \cup \{x\},B),e_1) \\
& & \quad V = V_1 \cup \{\mathsf{let} x = e_0 \mathsf{in} e_1)\} \\
& & \quad E = E_1 \cup \{(\mathsf{let} x = e_0 \mathsf{in} e_1, \mathsf{let} x = e_0 \mathsf{in} e_1, e_1)\} \\
& & \quad G' = \textbf{if } \mathsf{free}(e_1,x) = \{\} \textbf{ then } \{G_0\} \textbf{ else} \\
& & \qquad \{[G_0]e : e \in \mathsf{free}(e_1,x)\} \\
& & \quad V' = V \cup \bigcup\{V : (V,E) \in G'\} \\
& & \quad E' = E \cup \bigcup\{E : (V,E) \in G'\} \cup \\
& & \qquad \{(e, \mathsf{let} x = e_0 \mathsf{in} e_1, [e_0]e) : e \in \mathsf{free}(e_1,x)\} \\
& & \textbf{in} \\
& & \quad \mathsf{closeSet}((V',E'),\mathsf{free}(e_1,x),\mathsf{let} x = e_0 \mathsf{in} e_1)
\end{array}
$$

graph for the definition expression is copied and tagged with every instance of the bound identifier — unless there are no instances of the bound variable, in which case the graph is not altered.

Edges connect instances of x to the tagged expression vertices. The graph is closed below every instance of the bound identifier.

The pattern of recursion in the generation function is symmetric. The graphs for the two sides of an application expression are generated independently as in [McA98].

4.2 Reading Graphs

There are two distinct questions to be answered by reading a graph

1. Does the entire graph show that the expression is typeable?
2. What is the type represented by some vertex within the graph?

To find the typing, $\Gamma \vdash e : \tau$, for an expression, e, we must generate a graph, G, for the expression. Then find out whether the entire graph shows the expression is typeable (*i.e.* answer question 1), then if it is find the type τ represented by the vertex e (answer question 2).

We can consider question 1 to be analogous to type checking (it has a boolean response), and question 2 to be analogous to type inference (it results in an inferred type).

4.2.1 Type Checking

To check whether the entire graph represents a valid typing, we must visit every vertex and check that

- There are no branches, *i.e.* the vertex has at most one type constructor or one type variable vertex reachable from it (by search).
- The vertex is not part of a cycle (it is not reachable from itself). The cycle could involve type constructors and their connection points.
- The vertex is not the `unbound` type constructor.

These conditions are given by Definition 2.

Definition 2 (Valid Typing) (V, E) *is the graph for a correctly typed program iff it has*

- *No branches:*
$$\begin{aligned}&\nexists v \in V : \exists v_1, v_2 \in V : \\ &\quad v \mapsto^* v_1 \ \wedge \ (\nexists v_1' : v_1 \mapsto v_1') \ \wedge \\ &\quad v \mapsto^* v_2 \ \wedge \ (\nexists v_2' : v_2 \mapsto v_2') \ \wedge \\ &\quad v_1 \neq v_2\end{aligned}$$
Where $\mapsto^*$ is the reflexive transitive closure of $\mapsto$. i.e. there is at most one leaf reachable from any vertex (all type constructor and type variable vertices are leaves).

- *No cycles: $\nexists v \in V : v \Rightarrow^+ v$.*
Where $v \Rightarrow v'$ iff $v \mapsto v'$ or v' is a connection point of v, and $\Rightarrow^+$ is the non-reflexive transitive closure of $\Rightarrow$.

- *No unbound identifiers:* $\nexists i : \mathsf{unbound}_i \in V$.

An algorithm for checking this is a depth first search of the graph (branching at type constructors, to check the connection points). It stores the path used to reach the current vertex to detect cycles and also rejects the graph if any vertex has more than one type variable or constructor vertex as a descendant. For efficiency it is also convenient to build up a list of vertices already visited and know to be acceptable, this prevents areas of the graph being traversed more than once.

4.2.2 Type Inference

The search function seen earlier is also used to read graphs. Recall that the result of searching is a set of vertices without edges from them, and a set of type constructor vertices. There are three possibilities for the type of a vertex:

- If there is exactly one type constructor vertex and no other vertices, then the type is formed from that type constructor (the rest of the type can be built recursively),
- If there is a single other vertex (type variable, expression or connection point) then the type is a type variable identified by that vertex.
- otherwise there is no type (the graph represents some failure of type inference).

An algorithm for reading a type is in Figure 14, this takes a graph and vertex and produces a type. It will always terminate but will not give a type if there is any sort of conflict. See [McA99] for another type reading algorithm which will ignore unbound type constructors.

Figure 14 Algorithm for reading a type, takes a graph, a vertex, and a set of vertices already seen (initially empty); returns the type associated with the vertex (if one exists).

$$
\begin{array}{lcl}
\mathsf{type}((V,E),v,U) & = & \textbf{let} \\
& & \quad \textbf{if } v \in U \textbf{ then terminate (cyclic type)} \\
& & \textbf{case } \mathsf{search}((V,E),v) \\
& & \quad (\{\},\{\},\{v\}) \Rightarrow \mathsf{mkTyvar}(v) \\
& & \quad (\{\},\{\alpha\},\{\}) \Rightarrow \alpha \\
& & \quad (\{(\circ_1 \cdots \circ_n)c\},\{\},\{\}) \Rightarrow (\mathsf{type}((V,E),\circ_1)\cdots\mathsf{type}((V,E),\circ_n))c \\
& & \quad (cs,tvs,vs) \Rightarrow \textbf{terminate (conflict between constructors)}
\end{array}
$$

First search is used to find the significant vertices from the current vertex. If search finds only vertices with no children (and does not find any type constructors) then the type is a new type variable, and a relation is created relating this type variable to the set of vertices. If search finds only a single type constructor, then the type is formed by this constructor.

Function type checks for cycles and branching in the graph. This allows it to be used on graphs which do not represent valid typings (*i.e.* graphs which would be rejected by checkGraph). type alone is not sufficient, however, to see whether a graph represents a typing as it will not visit every vertex in the graph.

type does not tread the unbound type constructor specially. This allows it to produce types such as $\alpha \rightarrow$ unbound. As it stands, type is not suitable for reading the required types of unbound identifiers (such as I in Figure 7) as it will detect a conflict between unbound and the required type. It is clear that only a minor modification is required to ignore unbound, this modified algorithm is explored in [McA99]

5 Conclusions

We have seen a way of representing the results of type inference as a graph. Both typeable and untypeable programs can be represented in this way — this is the first work to treat typeable and untypeable programs equally. Algorithms for generating and reading graphs have been given. Another paper — [McA99] — shows that useful information can be extracted from graphs.

A Implementation

This appendix contains the key parts of SML source code for an implementation of the data structures and algorithms in this paper.

Source code files are available upon request (e-mail bjm@dcs.ed.ac.uk).

A.1 Abstract Syntax

A.1.1 Signature EXP

```
signature EXP =
  sig

    eqtype id

    val idToString : id -> string
    val idFromString : string -> id

    (* Every node of the abstract syntax tree has a number.
     These numbers are used to identify the program fragment which
     the node represents. *)
    eqtype number

    (* These imperative functions allow us to generate unique numbers *)
```

```
    val newNumber : unit -> number
    val reset : unit -> unit

    (* This value is a number which is never produced by newNumber.
       It is used for the expression vertices in graphs in the
       environment. *)
    val dummyNumber : number

    datatype exp = ID of id * number
                 | ABS of (id * exp) * number
                 | APP of (exp * exp) * number
                 | LET of (id * exp * exp) * number

    (* instances i e
     returns the numbers of all the fragments of e which are an instance
     of free identifier i. *)
    val instances : id -> exp -> number list

    (* Get the number of the top node in syntax tree. *)
    val number : exp -> number

    (* Extract a numbered fragment from an expression. *)
    exception NumNotFound
    val findFrag : (exp * number) -> exp

    (* Pretty printing *)
    val toString : exp -> string
    val numToString : number -> string

  end
```

A.2 Structure Exp : EXP

Implementation of this is trivial.

A.3 Representing Graphs

A.3.1 Signature GRAPH

```
signature GRAPH =
    sig

        (***** Data types for expressions and types *****)

        (* an identifier for program fragments *)
        eqtype fragment
```

```
eqtype expression

(* Identifiers for type constructors names *)
eqtype tycon
val funTycon : tycon
val unboundTycon : tycon
    (* Will update this to hold relevant fragment later *)
val newTycon : string * int -> tycon (* make new tycons *)

(***** Data types for vertices and edges *****)

(* There are two basic types of vertex *)
eqtype fragVertex

val getFrag : fragVertex -> fragment

(* Create an untagged fragment vertex from a fragment. *)
val quikTag : fragment -> fragVertex

eqtype tyconVertex

val getTycon : tyconVertex -> tycon
(* Is the vertex the UNBOUND tycon?  *)
val isUnbound : tyconVertex -> bool

eqtype tyvarVertex

(* Edges are connected to fragments, constructors,
 or connectors of constructors *)
datatype vertex = FRAG of fragVertex
                | TYCON of tyconVertex
                | CONNECTOR of tyconVertex * int
                | TYVAR of tyvarVertex

(* edges go between two vertices and are labeled with a fragment *)
eqtype edge
type edge_data = {v0: vertex, l: fragment, v1: vertex}
val edgeData : edge -> edge_data

(* Return the connection points of a tycon vertex *)
val conPts : tyconVertex -> vertex list

(***** Data types for graphs, and manipulation functions *****)

type graph
```

```
val empty : graph

(* Return list of all program fragment vertices *)
val vertices : graph -> vertex list

(* Get the kids (and corresponding edges) of a vertex *)
val kids : (graph * vertex) -> {v: vertex, e: edge} list

(* And the parents *)
val parents : (graph * vertex) -> {v: vertex, e: reverse_edge} list

(* Relable all edges in a graph with a given fragment
 Used when getting a graph from the environment. *)
val relEdges : fragment -> graph -> graph

(* Combine two graphs.
 If the fragment vertex sets are not disjoint then
 confusion will ensue. *)
val combine : graph * graph -> graph

(* Adding an fragVertex to a graph *)
val addExpVertex : graph -> fragment -> (graph * fragVertex)

(* Find the vertices for a particular fragment in a graph
 there may be more than one as they could be tagged. *)
val getExpVertices : graph -> fragment -> fragVertex list

(* Add an instance of a type constructor to a graph *)
val addTyconVertex : graph -> tycon -> (graph * tyconVertex)

(* Add a tyvar vertex (should not be used except to generate basis) *)
val addTyvarVertex : graph -> (graph * tyvarVertex)

(* Find important vertices from a root (used to generate types) *)
val search : ( vertex * graph ) ->
             ( tyconVertex list * tyvarVertex list * vertex list)

(* Adding an edge also closes the graph, this can add and
 remove tyvarVertices *)
val addEdge : ( edge_data * graph ) -> graph
val addEdgePrint : ( edge_data * graph ) -> graph (* For tracing *)

(* When we tag a graph, also provide a vertex, the tagged version
 of the vertex is returned.  Once a graph has been tagged, it
 should only be accessed through the single known vertex *)
```

```
    val tagGraph :
        (graph * fragVertex) -> fragVertex -> (graph * fragVertex)
  (*     tag me   and me          with me!       we're tagged now *)

    (***** toString *****)

    val tyconToString : tycon -> string

    val fVToString : fragVertex -> string

    val vToString : vertex -> string

    val eToString : edge -> string
    val revEToString : reverse_edge -> string
    val nondirEToString : nondir_edge -> string

    val toString : graph -> string

    val sizeToString : graph -> string

end
```

A.3.2 Functor GraphFun

The implementation of function `close` uses the fixed point combinator, as described in [McA97].

```
(* Functor for graphs *)

functor GraphFun(structure Exp : EXP)
    :> GRAPH where type fragment = Exp.number
             where type expression = Exp.exp =
    struct
        (***** Data types for expressions and types *****)

        (* an identifier for program fragments *)
        type fragment = Exp.number
        type expression = Exp.exp

        datatype tycon = UNBOUND (* of exp *) (* update later *)
                       | FUN
                       | CON of (string * int)
                         (* the int is the arity *)
        val funTycon = FUN
        val unboundTycon = UNBOUND
        val newTycon = CON
```

```
fun arity UNBOUND = 0
  | arity FUN = 2
  | arity (CON (_, i)) = i

(***** Data types for vertices and edges *****)

(* Fragments can be tagged *)
datatype fragVertex = TAGGED of fragVertex list * fragment

fun quikTag e = TAGGED([], e)

fun getFrag (TAGGED (_, f)) = f
fun getTag (TAGGED (t, f)) = t

type tyconVertex = (int * tycon)

fun getTycon (_, c) = c

(* Just a unique identifier for tyvars *)
type tyvarVertex = int

datatype vertex =
    FRAG of fragVertex
  | TYCON of tyconVertex
  | CONNECTOR of tyconVertex * int
  | TYVAR of tyvarVertex

(* To get a list of connection points for a tyconVertex *)
fun conPts (i, c) =
    List.tabulate (arity c, fn n => CONNECTOR((i, c), n))

fun isUnbound (_, s) = s=unboundTycon

type edge = {v0: vertex, l: fragment, v1: vertex, exp: bool}
type edge_data = {v0: vertex, l: fragment, v1: vertex}
fun edgeData (e : edge) = {v0= #v0 e, l= #l e, v1= #v1 e}

(***** Data types for graphs, and manipulation functions *****)

(* graphs include a number (max) used to give instances of type
 constructors and type variables unique identifiers *)
type graph = {V : vertex list, E : edge list,
              max : int}

val empty = {V = [], E = [], max = 0}
```

```
fun vertices ({V, ...}:graph) = V

fun relEdges e {V,E,max} =
    {V=V,
     E=map (fn {v0,l,v1,exp}=>{v0=v0,l=e,v1=v1, exp=exp}) E,
     max=max}

(* None of the graph manipulation functions check for vertex or edge
 existance. *)

fun combine ({V = V0, E = E0, max = m0},
             {V = V1, E = E1, max = m1}) =
    let
        (* renumber identifiers on tycons and tyvars, so there are no
         clashes. *)
        fun reV (TYCON(i, c)) = TYCON(i + m0, c)
          | reV (CONNECTOR((i, c), ind)) = CONNECTOR((i+m0, c), ind)
          | reV (TYVAR i) = TYVAR(i + m0)
          | reV v = v
        fun reE {v0, l, v1, exp} =
            {v0 = reV v0, l=l, v1=reV v1, exp = exp}
    in
        {V = V0 @ (map reV V1),
         E = E0 @ (map reE E1),
         max = m0 + m1}
    end

(* It would be possible to add a tyvar for every new fragment,
 but I just let the fragment vertex serve as a tyvar *)
fun addExpVertex {V, E, max} e =
    ({V = (FRAG (quikTag e))::V,
      E = E,
      max = max},
     quikTag e)

(* Function for getting all vertices which are tagged versions
 of some fragment *)
fun getExpVertices ({V,...}:graph) e =
        List.mapPartial
        (fn FRAG(TAGGED(t, e')) => if e=e' then
                                       SOME(TAGGED(t, e'))
                                   else NONE
          | _ => NONE)
         V
```

```
(* We could add a tyvar vertex for every connection point, but
 we just let the connection points serve as tyvars *)
fun addTyconVertex {V, E, max} c =
    ({V = (TYCON(max+1, c))::V,
      E = E,
      max = max + 1},
     (max+1, c))

(* Add a tyvar vertex *)
fun addTyvarVertex {V, E, max} =
    ({V = (TYVAR(max+1))::V, E = E, max = max + 1}, max + 1)

(* Find all the immediate descendants of a vertex
 (Used internally to close graphs) *)
fun kids ({E, ...}:graph, v) =
    map (fn e as {v1, ...} => {v=v1, e=e})
    (List.filter (fn {v0, ...} => v0=v) E)

val kids' = (map #v) o kids

fun parents ({E, ...}:graph, v) =
    map (fn e as {v0, ...} => {v=v0, e=e})
    (List.filter (fn {v1, ...} => v1=v) E)

(* Used to prevent duplicate entries when building up lists of
 reachable vertices *)
fun union l1 l2 =
    l1 @ (List.filter (fn i => not (List.exists (fn i' =>i'=i) l1)) l2)

fun removeTyvars tvs =
    List.filter (fn v => not (List.exists (fn i => TYVAR i=v) tvs))

fun removeCons cs =
    List.filter (fn v => not (List.exists (fn v' => TYCON v'=v) cs))

(***** toString *****)

fun tyvarToString i = "'" ^ (Int.toString i)

fun tyconToString UNBOUND = "UNBOUND"
  | tyconToString FUN = "->"
  | tyconToString (CON(s, i)) = s

fun doC (id, UNBOUND) = "(UNBOUND #"^(Int.toString id)^")"
```

```
  | doC (id, FUN) =  "(-> #"^(Int.toString id)^")"
  | doC (id, CON (c, i)) = "("^c^" #"^(Int.toString id)^")"

fun doT([], e) = Exp.numToString e
  | doT((TAGGED(t, e))::T, E) =
    "["^(doT (T, E))^"]"^(doT (t, e))
and fVToString (TAGGED f) = doT f
and vToString (FRAG f) = fVToString f
  | vToString (CONNECTOR (id, i)) =
    (doC id)^"."^(Int.toString i)
  | vToString (TYCON id) =
    doC id
  | vToString (TYVAR i) = tyvarToString i

fun eToString ({v0, l, v1, ...} : edge) =
    (vToString v0) ^ "  |- " ^
    (Exp.numToString l) ^ " ->  "^
    (vToString v1)

fun revEToString ({v0, l, v1, ...} : reverse_edge)=
    (vToString v1) ^ "  <- " ^
    (Exp.numToString l) ^ " -|  "^
    (vToString v0)

fun nondirEToString (FORWARD e) = eToString e
  | nondirEToString (REVERSE e) = revEToString e

fun toString ({E, V, max}) =
    let
        fun doVar v = "'"^(Int.toString v)


        val Vstring =
            List.foldl (fn (v, s) => s^"\n  "^(vToString v) )
            "Vertices:" V

        val Estring =
            List.foldl (fn (e, s) => s^"\n  "^(eToString e) )
            "Edges:" E
    in
        (* Vstring ^ "\n" ^ *)
        Estring ^ "\n"
        (* ^ "max = "^(Int.toString max)^"\n" *)
    end
```

```
fun sizeToString ({E, V, ...} : graph) =
    "|V| = " ^ (Int.toString (length V)) ^
    " |E| = " ^(Int.toString (length E))


(**** IMPORTANT FUNCTION: search ****)

(* Collect all leaves reachable from a vertex
 split into tycons, tyvars and others (fragments and connections) *)
fun search ((TYCON c), G : graph) = ([c], [], [])
  | search ((TYVAR v), G) = ([], [v], [])
  | search (v, G) =
    (case kids' (G, v) of
         [] => ([], [], [v])
       | ks =>
             List.foldl
             (fn (k, (cs, tvs, vs)) =>
              let
                  val (cs', tvs', vs') = search (k, G)
              in
                  (union cs' cs,
                   union tvs' tvs,
                   union vs' vs)
              end)
             ([], [], []) ks )


(**** IMPORTANT FUNCTION: close ****)

(* Close has to take a list of previously closed vertices to
 prevent looping *)

(* This function has been written for use with the
 fixed point combinator.  See [McA97] for information on
 this programming style. *)

fun close_ close (G : graph, TYCON c, lab, seen) =
    if List.exists (fn v' => v'=TYCON c) seen then
        (G, seen)
    else
        let
            val cPs = conPts c
        in
            List.foldl
```

```
          (fn (v, (G', seen')) => close (G', v, lab, seen'))
          (G, seen) cPs
      end
  | close_ close (G : graph as {V, E, max}, v, lab, seen) =
    if List.exists (fn v' => v'=v) seen then
         (G, seen)
    else
        (case search (v, G) of
             ([], [], [v']) => (G, seen)
           | ([], [v'], []) => (G, seen)
           | ([v'], [], []) => (G, seen)
           (* There is only one vertex, no closing necessary. *)
           | ([], [], vs) =>
                 (* There are a bunch of miscellaneous vertices.
                  Create a tyvar and link all the vertices to it. *)
                 let
                     val alpha = TYVAR (max + 1)
                 in
                     ({V=alpha::V,
                       max = max + 1,
                       E =
                        (map
                         (fn v' => {v0=v', l=lab, v1=alpha, exp=false})
                        vs) @ E},
                      seen)
                 end
           | ([], alpha::tvs, vs) =>
                 (* There are is at least one tyvar, and
                  possibly some assorted other vertices.
                  Merge all the tyvars into one, and connect
                  all other vertices to the remaining tyvar.*)
                 let
                     (* Change edges of members of tvs
                      to go to alpha *)
                     val E' = map
                          (fn e as {v0, l, v1, exp} =>
                           if List.exists (fn i => TYVAR i=v1) tvs then
                                {v0=v0, l=l, v1=TYVAR alpha, exp=exp}
                           else
                                 e )
                         E

                     (* Add edges from vertices in vs to alpha *)
                     val E'' =
                         map (fn v =>
```

```
                    {v0=v, l=lab, v1=TYVAR alpha, exp=false})
              vs
        in
            ({V = removeTyvars tvs V,
              E = E' @ E'',
              max = max},
             seen)
        end
  | (cs, tvs, vs) =>
        (* There are some constructors, possibly some tyvars,
           and possibly some other vertices.
           Merge like constructors,
           Merge each tyvar with every remaining constructor,
           Connect other vertices to every remaining
           constructor.*)
        let
            (* Go through cs merging like constructors,
             return remaining constructors and new graph *)
            fun merger ((G, seen), []) = ([], (G, seen))
              | merger (({V, E, max}:graph, seen),
                        ((id, c)::cs')) =
                let

                    (* Find remaining constructors like
                     the current one and unlike it *)
                    val (likeConstructors, unlikeConstructors)
                        =
                        List.partition
                        (fn (id', c') => c'=c)
                        cs'


                    (* remove like constructors from V *)
                    val V' = removeCons likeConstructors V

                    (* a function to change vertices in
                     likeConstructors into the current vertex
                     and leave other vertices unchanged *)
                    fun changeV (TYCON v) =
                        if List.exists
                            (fn v'=> v=v')
                            likeConstructors then
                            TYCON (id, c) else TYCON v
                      | changeV (CONNECTOR(v, i)) =
                        if List.exists
```

```
                  (fn v'=> v=v')
                  likeConstructors then
                  CONNECTOR((id, c), i) else
                  CONNECTOR(v, i)
              | changeV v = v
            (* Move edges from like constructors to
             current one *)
            val E' = map
                (fn {v0, l, v1, exp} =>
                 {v0=changeV v0, l=l,
                  v1=changeV v1, exp=exp})
                E
            val (G', seen') =
                close({V=V', E=E', max=max},
                      TYCON(id, c), lab,
                      v::seen)

            val (cs', (G'', seen'')) =
                merger((G', seen'),
                       unlikeConstructors)
        in
            ((id, c)::cs',
             (G'', (TYCON (id, c))::seen''))
        end (* of merger *)

    val (cs', ({V, E, max}, seen')) =
        merger ((G, seen), cs)

    (* Next, merge each tyvar with _every_
     remaining con.
     This means removing the tyvars from vertex set,
     and replacing edges to the tyvars with sets of
         edges to the constructors. *)
    val noTyvarsV = removeTyvars tvs V

    fun doE [] = []
      | doE ((e as {v0, l, v1, exp})::E) =
        (if List.exists (fn v => TYVAR v = v1) tvs
         then
             map
             (fn c =>
              {v0=v0, l=lab, v1=TYCON c, exp=false})
             cs'
         else
             [e]) @
```

```
          (doE E)

val sortedTyvarsE : edge list= doE E

(* Now, if there is an UNBOUND tycon vertex,
 find its `owners' and connect these to every
 other tycon *)

val sortedUnbounds : edge list =
    let
        val (unbounds, others) =
            List.partition
            (fn v => isUnbound v)
            cs'
        fun getOwners u =
            map #v0
            (List.filter
             (fn {v0=FRAG v0, l, v1, ...} =>
                  getFrag v0 = l
                  andalso
                  v1 = TYCON u
               | _ => false)
             sortedTyvarsE)
        fun doU u (FRAG v) =
            map
            (fn c =>
             {v0=FRAG v, l=getFrag v, v1=TYCON c,
              exp = false})
            others
          | doU _ _ = raise Fail "wagga wagga"
    in
        case unbounds of
            [u] =>
                List.concat
                (map (doU u) (getOwners u))
          | _ => []
    end

(* Finally, connect up elements of vs to cs' *)
val sortedOthersE : edge list =
    List.concat
    (List.map
     (fn c =>
      List.map
       (fn v =>
```

```
                                {v0=v, l=lab, v1=TYCON c, exp = false}) vs)
                              cs')

                      in
                          ({V = noTyvarsV,
                            E = sortedTyvarsE @
                                sortedUnbounds @
                                sortedOthersE,
                            max = max}, seen')
                      end
                 ) (* End case search (G, v) *)
                 (* End fun close_ *)

    fun FIX f_ x = f_ (FIX f_) x

    fun close (G, v, lab) =
        let
            val result = FIX close_ (G, v, lab, [])
        in
            #1 result
        end

    (* When adding an edge, close below the start vertex *)
    fun addEdge ({v0, l, v1}, {V, E, max}) =
            close({V=V,
                   E = {v0=v0, l=l, v1=v1, exp = true}::E,
                   max = max},
                  v0, l)

    fun tagGraph ({V, E, max}, TAGGED(l, e)) t =
        let
            fun tagV (FRAG(TAGGED(l, e))) = FRAG(TAGGED(t::l, e))
              | tagV v = v

            fun tagE {v0, l, v1, exp} =
                {v0=tagV v0, l=l, v1=tagV v1, exp = exp}
        in
            ({V = map tagV V,
              E = map tagE E,
              max = max}:graph,
             TAGGED(t::l, e):fragVertex)
        end


end (* of functor GraphFun *)
```

A.3.3 Structure Graph : GRAPH

```
structure Graph = GraphFun(structure Exp = Exp)
```

A.4 Environment

A.4.1 Signature ENV

```
signature ENV =
    sig

        type id (* Syntactic identifier *)
        type graph (* Graphs *)
        eqtype vertex (* vertices *)
        type env (* Environments *)

        exception OutOfScope

        (* lookup returns either nothing (in scope, no type) a graph
        and vertex (in scope, with type) or raises exception
        OutOfScope *)
        val lookup : env -> id -> (graph * vertex) option

        (* update removes any existing entry and adds empty entry *)
        val update : env -> id -> env

        (* Basis environment *)
        val basis : env

    end
```

A.4.2 Structure Env : ENV

This is trivial to implement.

A.5 Graph Generation

A.5.1 Signature GENERATE

```
signature GENERATE =
    sig

        eqtype exp

        type env

        type graph
```

```
    eqtype vertex

    val generate : env -> exp -> (graph * vertex)

end
```

A.5.2 Functor GenerateFun

```
functor GenerateFun (structure Exp : EXP
                     structure Graph : GRAPH
                     structure Env : ENV
                     sharing type Exp.number = Graph.fragment
                     sharing type Env.id = Exp.id
                     sharing type Env.graph = Graph.graph
                     sharing type Env.vertex = Graph.vertex)
    :> GENERATE
       where type exp = Exp.exp
       where type graph = Graph.graph
       where type vertex = Graph.fragVertex
       where type env = Env.env =
    struct

        type exp = Exp.exp

        type env = Env.env

        type graph = Graph.graph
        type vertex = Graph.fragVertex

        fun generate gamma (Exp.ID(i, n)) =
            ((case Env.lookup gamma i of
                   SOME (G, v) =>
                        (* From the basis *)
                        let
                            val (G', vId) = Graph.addExpVertex G n
                            val G'' = Graph.relEdges n G'
                        in
                            (Graph.addEdge ({v0=Graph.FRAG vId, l=n, v1=v}, G''),
                             vId)
                        end
                 | NONE => (* Just create a unary graph *)
                        Graph.addExpVertex Graph.empty n)
                handle Env.OutOfScope =>
                    (* The exception means i is unbound *)
                    let
                        val (G, vUnb) =
```

```
                Graph.addTyconVertex Graph.empty Graph.unboundTycon
            val (G', vId) = Graph.addExpVertex G n
        in
            (Graph.addEdge
               ({v0=Graph.FRAG vId, l=n, v1=Graph.TYCON vUnb}, G'),
             vId)
        end)
  | generate gamma (Exp.ABS((i, e0), n)) =
    let

        val gamma' = Env.update gamma i

        (* Graph for subexpression *)
        val (G0, v0) = generate gamma' e0

        (* Find vertex for each instance of formal parameter *)
        val instances =
            List.concat
            (map (Graph.getExpVertices G0) (Exp.instances i e0))

        (* Add vertex for abstraction expression *)
        val (G', v) = Graph.addExpVertex G0 n

        (* Add vertex for function type constructor *)
        val (G'', vTycon) = Graph.addTyconVertex G' Graph.funTycon

        (* Edges:
             abstraction expression |-> function type constructor
             Right of tycon |-> body expression
             Left of tycon |-> every instance of param
         *)
        val newE =
            {v0=Graph.FRAG v, l=n, v1=Graph.TYCON vTycon}::
            {v0=Graph.CONNECTOR (vTycon, 1), l=n, v1=Graph.FRAG v0}::
            (map
             (fn n' =>
              {v0=Graph.CONNECTOR (vTycon, 0), l=n, v1=Graph.FRAG n'})
             instances)

    in
        (* Finally, add all the edges *)
        (List.foldl Graph.addEdge G'' newE, v)
    end
  | generate gamma (Exp.APP((e0, e1), n)) =
    let
```

```
        (* Graphs for subexpressions *)
        val (G0, v0) = generate gamma e0
        val (G1, v1) = generate gamma e1

        (* Combine, and add vertex for application *)
        val (G, v) = Graph.addExpVertex (Graph.combine (G0, G1)) n

        (* Add vertex for fun type constructor *)
        val (G', vTycon) = Graph.addTyconVertex G Graph.funTycon

        (* Edges:
             app exp |-> right of tycon
             left of tycon |-> e1 (it must be a parameter)
             e0 |-> tycon (it must be a function
         *)
        val newE =
            [{v0=Graph.FRAG v, l=n, v1=Graph.CONNECTOR (vTycon, 1)},
             {v0=Graph.CONNECTOR(vTycon, 0), l=n, v1=Graph.FRAG v1},
             {v0=Graph.FRAG v0, l=n, v1=Graph.TYCON vTycon}]

        val result =
            (* Finally, add edges *)
            (List.foldl Graph.addEdge G' newE,
             v)
    in
        result
    end
  | generate gamma (Exp.LET((i, e0, e1), n)) =
    let
        val gamma' = Env.update gamma i

        (* Graphs for subexpression *)
        val (G0, v0) = generate gamma e0
        val (G1, v1) = generate gamma' e1

        (* Add vertex for declaration expression *)
        val (G, v) = Graph.addExpVertex G1 n

        (* Add edge from declaration to the use expression (e1) *)
        val G' =
            Graph.addEdge ({v0=Graph.FRAG v, l=n, v1=Graph.FRAG v1}, G)

        val ins = Exp.instances i e1

        (* Find every instance of bound identifier *)
```

```
                val instances =
                    List.concat
                    (List.map (Graph.getExpVertices G') (Exp.instances i e1))

                (* Function to connect every instance of bound identifier
                 to a copy of the definition expression *)
                fun addInst (vId, G) =
                    let
                        (* Create copy of definition *)
                        val (Gi, vDef) = Graph.tagGraph (G0, v0) vId

                        (* Find vertex for instance of bound id *)
                        (* val vId = Graph.getExpVertex G i *)

                        (* Put the two graphs together *)
                        val G' = Graph.combine (G, Gi)
                    in
                        (* Add edge from id to def *)
                        Graph.addEdge
                        ({v0=Graph.FRAG vId, l=n, v1=Graph.FRAG vDef}, G')
                    end
            in
                (* Do every instance of bound id *)
                (List.foldl addInst G' instances, v)
            end

    end (* of function GenerateFun *)
```

A.5.3 Structure Generate : GENERATE

```
structure Generate = GenerateFun(structure Exp = Exp
                                 structure Graph = Graph
                                 structure Env = Env)
```

A.6 Reading Graphs

A.6.1 Signature READ_GRAPHS

```
signature READ_GRAPHS =
    sig

        type graph
        and vertex
        and fragVertex
```

```
        and nondir_edge

        eqtype tyvar
        val mkTyvar : string -> tyvar (* only use to make basis! *)
        eqtype tycon
        datatype ty = VAR of tyvar
                    | FUN of ty * ty
                    | UNBOUND
                    | CON of ty list * tycon

        val stringTy : ty -> string
        val size : ty -> int

        type fragment

        (* BASIC ROUTINES *)

        val checkGraph : graph -> bool

        val readType : graph * vertex -> ty


    end
```

A.6.2 Functor ReadGraphsFun

The implementation of function `readType` uses the fixed point combinator, as described in [McA97].

```
functor ReadGraphsFun(structure Graph : GRAPH) :>
    READ_GRAPHS
    where type graph = Graph.graph
    where type vertex = Graph.vertex
    where type fragVertex = Graph.fragVertex
    where type tycon = Graph.tycon
    where type fragment = Graph.fragment
    where type expression = Graph.expression
    where type nondir_edge = Graph.nondir_edge =
    struct

        type graph = Graph.graph
        and vertex = Graph.vertex
        and fragVertex = Graph.fragVertex
        and nondir_edge = Graph.nondir_edge

        datatype tyvar = VERT_TV of Graph.vertex
                       | NAMED_TV of string
```

```
(* Tyvars are usually a tyvarVertex, expVertex or connection point.
 we also use named tyvars while generating basis, but they disappear
 quickly (they are a HACK and will be changed to special tyvars
 at a later date). *)

val mkTyvar = NAMED_TV

type tycon = Graph.tycon

datatype ty = VAR of tyvar
             | FUN of ty * ty
             | UNBOUND (* Special pseudo type constructor *)
             | CON of ty list * tycon

fun size (VAR _) = 1
  | size (FUN (t1, t2)) = 1 + (size t1) + (size t2)
  | size UNBOUND = 1
  | size (CON (l, _)) = List.foldl (fn (ty, t) => t + (size ty)) 1 l

type fragment = Graph.fragment
type expression = Graph.expression

(*******************************************************)

fun stringTy (VAR (VERT_TV tyvar)) =
    " '" ^ (Graph.vToString tyvar) ^ " "
  | stringTy (VAR (NAMED_TV tyvar)) = "'"^tyvar
  | stringTy (FUN(ty1, ty2)) =
    "( "^(stringTy ty1)^" ) -> ( "^(stringTy ty2)^" )"
  | stringTy UNBOUND = "UNBOUND"
  | stringTy (CON(l, tycon)) =
    case l of
        [] => Graph.tyconToString tycon
      | [ty] =>  (stringTy ty)^" "^(Graph.tyconToString tycon)
      | l => "(" ^ (Pretty.commaSep (map stringTy l)) ^ ") " ^
             (Graph.tyconToString tycon)

fun stringRel r =
    Pretty.commaSep
     (map
      (fn (a, V)=>
       "(" ^ a ^ ", (" ^
       (Pretty.commaSep (map Graph.vToString V) ^ "))" )  )
      r)
```

```
(**************************************************)

val member = fn x => List.exists (fn x' => x=x')

exception Cycle of Graph.vertex
exception Conflict of Graph.vertex
exception Unbound of Graph.vertex

(* Basic operation of checkTree:
    Given vertex, v, path used to reach v and list of ok vertices.
    If v is ok then return list of ok vertices (pass)
    If v is already on the path then fail
    Find v's kids, if there is only one and it has no kids
   then return v::ok vertices (pass)
    If there is only one and it is a constructor, call
   checkForest on the connection points to get a new set of ok
   vertices.  Return v::new ok vertices.
    If there is more than one kid, fail.
  checkForest:
    fold across a list of vertices, building up ok vertices *)

fun checkGraph G =
    let
        fun checkTree path (v, okVs) =
            if member v okVs then
                okVs
            else

                if member v path then
                    raise Cycle v
                else
                    let
                        val kids =
                            case Graph.search (v, G) of
                                ([], [], []) => []
                              (* no kids *)
                              | ([], [], [v]) => []
                              (* one kid *)
                              | ([], [alpha], []) => []
                              (* one tyvar kid *)
                              | ([con], [], []) =>
                                   if Graph.isUnbound con then
                                        raise Unbound v
                              (* The vertex attatched to this
                               exception is not necessarily the
```

```
                                actual unbound identifier (arse!). *)
                                     else
                                         Graph.conPts con
                               (* Constructor kid:
                                Check connection points *)
                               | _ => raise Conflict v
                    (* multiple kids *)
                    in
                        v::(checkForest (v::path) okVs kids)
                    end

            and checkForest path =
                List.foldl (checkTree path)

            val print' = fn _ => () (* Or replace by 'print.' *)

        in
            (checkForest [] []
             (List.filter (fn Graph.FRAG _ => true | _ => false)
              (Graph.vertices G));
             true)
              (* Compiler warning from the discarded result! *)
            handle Cycle v =>
                       (print' ("Cycle at "^(Graph.vToString v)^"\n");
                        false)
                 | Conflict v =>
                       (print' ("Conflict at "^(Graph.vToString v)^"\n");
                        false)
                 | Unbound v =>
                       (print' ("Unbound id at "^(Graph.vToString v)^"\n");
                        false)
        end (* of fun checkGraph *)


    (*******************************************************)

    (* Type will ignore 'unbound' unless there are no alternatives,
       this allows it to be used to read the required types of
       unbound identifiers. *)


    fun readType (G, v) =
        let

(* This function has been written for use with the
```

```
fixed point combinator.  See [McA97] for information on
this programming style. *)

            fun rT_ (rT : graph * vertex * vertex list -> ty )
                  (G, v, seen) =
                if (List.exists (fn v' => v'=v) seen) then
                    raise Cycle v
                else
                    let
                        val (cs, vs, tvs) = Graph.search(v, G)
                            (* Remove all instances of the unbound
                             type constructor, unless this was
                             the only thing returned *)
                        val cs' =
                            (case cs of
                                 [c] => if Graph.isUnbound c
                                           andalso (vs, tvs) <> ([], [])
                                        then [] else [c]
                               | cs =>
                                     List.filter
                                     (not o Graph.isUnbound) cs)
                    in
                        case (cs', vs, tvs) of
                            ([], [], [v]) =>
                                (* Return a tyvar *)
                                VAR (VERT_TV v)
                          | ([], [v], []) =>
                                VAR (VERT_TV (Graph.TYVAR v))
                          | ([v], [], []) =>
                                (* Unique type constructor,
                                 return a type.  *)
                                let
                                    val c = Graph.getTycon v
                                    val seen' = (Graph.TYCON v)::seen
                                (* Assume c has arity 2 (Bad Hack!) *)
                                in
                                    if c = Graph.funTycon then
                                        FUN(rT(G,
                                               Graph.CONNECTOR(v, 0),
                                               seen'),
                                            rT(G,
                                               Graph.CONNECTOR(v, 1),
                                               seen'))
                                    else
                                        if c = Graph.unboundTycon then
```

```
                                        UNBOUND
                                    else
                                        let
                                            val tys =
                                                map
                                                (fn v =>
                                                 rT(G, v, seen'))
                                                (Graph.conPts v)
                                        in
                                            CON (tys, c)
                                        end
                            end
                      | _ =>
                            (* There must be more than one vertex *)
                            raise Conflict v
                end

            fun FIX f x = f (FIX f) x

            fun wrap rT_ rT (G, v, seen) =
                let
                    val (ty) = rT_ rT (G, v, seen)
                    val _ =
                        print
                        ("\nvertex " ^ (Graph.vToString v) ^
                         "\nhas type " ^ (stringTy ty))
                in
                    ty
                end

            val rT_' = wrap rT_

            val rT = FIX rT_

        in
            rT (G, v, [])
        end (* of fun readType *)


    end
```

A.6.3 Structure ReadGraphs : READ_GRAPHS

```
structure ReadGraphs = ReadGraphsFun(structure Graph = Graph);
```

A.7 Testing

```
val e = (* Example expression *) ;
val (G, v) = Generate.generate Env.basis e ;
val ok = ReadGraphs.checkGraph G ; (* true if e is typeable in Env.basis *)
val t = ReadGraphs.readType (G, Graph.FRAG v)
```

References

[BS93] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on programming languages and systems*, 2(1):17–30, March 1993.

[BS95] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, Computer Science Department, November 1995. `http://www.cs.sunysb.edu/~stark/REPORTS/INDEX.html`.

[DB96] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, (27):37–83, 1996.

[Dug98] Dominic Duggan. Correct type explanation. In *Workshop on ML*, pages 49–58. ACM SIGPLAN, 1998.

[JW86] Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type-inference. In *ACM Symposium on Principles of Programming Languages*, number 13, pages 44–57. ACM, ACM Press, 1986.

[Ler99] Xavier Leroy. Re: Type inference algorithms. `news:comp.lang.ml`, January 1999. Archived at `ftp://pop.cs.cmu.edu/usr/rowan/sml-archive/sml-archive.99`, message ID `lyzu2xcoi17.fsf@estephe.inria.fr`.

[LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[McA97] Bruce J. McAdam. That about wraps it up — Using FIX to handle errors without exceptions, and other programming tricks. Technical Report ECS–LFCS–97–375, Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, UK, November 1997.

[McA98] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.

[McA99] Bruce J. McAdam. Generalising techniques for type explanation. In Greg Michaelson and Phil Trinder, editors, *Trends in Functional Programming 1999 (Proceedings of Scottish Functional Programming Workshop)*. Intellect, 1999. To appear.

[SA93] Zhong Shao and Andrew Appel. Smartest recompilation. In *ACM Symposium on Principles of Programming Languages*, number 20. ACM, ACM Press, January 1993.

[Soo90] Helen Soosaipillai. An explanation based polymorphic type-checker for Standard ML. Master's thesis, Department of Computer Science, Heriot-Watt University, 1990.

[Tur90] David A. Turner. Enhanced error handling for ML. CS4 Project, Department of Computer Science, The University of Edinburgh, May 1990.

[Wan86] Mitchell Wand. Finding the source of type errors. In *ACM Symposium on Principles of Programming Languages*, number 13, pages 38–43. ACM, ACM Press, 1986.

[Yi99] Kwangkeun Yi. Re: Type inference algorithms. `news:comp.lang.ml`, January 1999. Archived at `ftp://pop.cs.cmu.edu/usr/rowan/sml-archive/sml-archive.99`, message ID `m1k8y99bwr.fsf@compiler.kaist.ac.kr`.