Visual Type Inference

Martin Erwig
School of EECS
Oregon State University
Corvallis, OR 97331, USA
erwig@eecs.oregonstate.edu

January 12, 2005

Abstract

We describe a type-inference algorithm that is based on labeling nodes with type information in a graph that represents type constraints. This algorithm produces the same results as the famous algorithm of Milner, but is much simpler to use, which is of importance especially for teaching type systems and type inference.

The proposed algorithm employs a more concise notation and yields inferences that are shorter than applications of the traditional algorithm. Simplifications result, in particular, from three facts: (1) We do not have to maintain an explicit type environment throughout the algorithm because the type environment is represented implicitly through node labels. (2) The use of unification is simplified through label propagation along graph edges. (3) The typing decisions in our algorithm are dependency-driven (and not syntax-directed), which reduces notational overhead and bookkeeping.

Keywords: Type inference algorithm, lambda calculus, polymorphic type system, graph

1 Introduction

Type inference is an essential part of statically typed functional programming languages, such as Haskell or ML. The process of type inference is not just a "problem" of the compiler in the sense that once an algorithm is found and implemented the "problem" can be regarded as being solved. Knowledge about how type inference works is an important skill that is required to use a strongly typed language effectively. For example, it is almost impossible to understand type error messages—an important guidance in writing correct programs—without knowing about type inference. Similarly, the annotation of functions with polymorphic types can be performed successfully only with a certain understanding of a language's type system, which defines how types are related to syntactic constructs of the language.

The Hindley/Milner type system [15] and the type-inference algorithm $\mathcal{W}$ is the basis for most statically typed functional languages. Although the algorithm is not very complex and is fairly easy to implement, it is difficult to apply "by hand", which is needed sometimes to understand the reasoning of the type checker or to just think about complex type situations.

Consider as an example the inference of the type for the function composition combinator $\lambda f.\lambda g.\lambda x.f\ (g\ x)$. The application of the algorithm to this example is quite tedious; see the Appendix. It is the heavy use of unification together with the need to maintain type assumptions under substitutions that makes the algorithm difficult to apply by hand. These difficulties are aggravated by the recursive nature of the algorithm, which forces one to frequently invent and rename variables, keep track of local bindings, etc. The complexity of demonstrating how the algorithm works for examples is probably also the reason why type inference is mostly explained in different ways, see for example, [20, 16], which presents an unfortunate situation from a teaching point of view.

In this paper we propose an alternative algorithm $\mathcal{G}$ that is based on labeling nodes in a graph. This algorithm produces the same results as $\mathcal{W}$, but it is much simpler to use. In particular, no type environment is required, and unification is performed mostly implicitly through a graph labeling technique that simulates the flow of types through the graph. More specifically, the presented algorithm is based on the representation of lambda expressions by graphs that specify their types. Nodes in these type graphs are labeled with type expressions, and edges represent constraints between types. These constraints are solved by letting types flow along these edges and sometimes also by restructuring graphs. The type-inference process consists of two steps. First, we have to create a so-called *type-flow graph* that serves as a specification of the type to be inferred. Then in a second step we perform operations on this graph that leaves as a result the inferred type as a node label (or reveals a type error).

Let us consider as an example the composition combinator. The type of a lambda abstraction is a function type. In this example it is a function type of three parameters. We create a node that is labeled with such a function type, however we leave the parameter types and the result type unspecified, which is represented by using unlabeled nodes in the type expression.

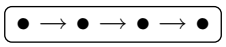

Figure 1: Node labeled with type expression.

Figure 1 shows a graph that consists of five nodes, four of which are unlabeled (represented by •) and one of which is labeled with the type expression $\bullet \to \bullet \to \bullet \to \bullet$. Note that in this representation type expressions can contain nodes. In general, unlabeled nodes are visually represented by •, but to emphasize their relationships to the variables from the function definition, we may also use the variable names of the parameters instead.

The result type of the function is determined by the result type of the body of the lambda abstraction, $f\ (g\ x)$, which is represented in the graph by a directed edge from the result node of the application to the result node of the function type. In each type-flow graph we designate a node that is going to be labeled by the result type, and we call that node the *result node*. In our example, the result node is the node labeled with the function type.

The type of an application is given by the result type of the function being applied, which must be, of course, a function type. In addition, the parameter type of the function must agree with

the type of the argument. These requirements can be represented in a graph by creating a node labeled with a function type, adding an edge between this node with the node that represents the function in the graph, and connecting the node representing the type of the argument with the parameter type of the function type. In our example we therefore create two nodes labeled $\bullet \to \bullet$ for the applications of g to x and f to $g\ x$. These nodes are connected to f and g representing the fact that f and g must be function types (since they are applied). Moreover, the argument type of g's function type is connected to x, and g's result type is connected to the argument type of f's function type. Finally, the result type of f's function type, which represents the result type of the function body, is connected to the result node of the composition combinator.[1]

As a result we obtain the type-flow graph shown in Figure 2. In the following we refer to the unnamed, unlabeled nodes in the graph from top to bottom, left to right by r, f_1, f_2, g_1, and g_2, respectively.

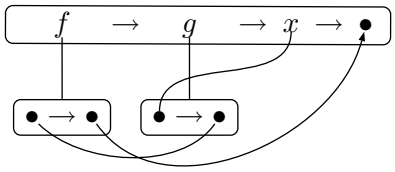

Figure 2: Type-flow graph for the composition combinator.

To infer the type for this type-flow graph we start by labeling all nodes that do not depend on other nodes by fresh type variables. For example, unlabeled nodes that do not have predecessors in the graph are independent. In the above graph, we find that the node f_2 has this property, so we can label the node with a. (Note that an undirected edge corresponds to two directed edges in either direction so that any node that is attached to an undirected edge always has a predecessor.) Moreover, all unlabeled nodes that are transitively connected through undirected edges to other unlabeled nodes such that no node in such a group has other labeled predecessors are also considered independent and can be labeled by a fresh type variable. We can assign one and the same fresh type variable to all nodes in such a group. In the example, we observe two groups of nodes, namely $\{f_1, g_2\}$ and $\{g_1, x\}$, and we can assign to them the type variables b and c, respectively.

The edges in a type-flow graph represent constraints on the labels of the connected nodes: An undirected edge expresses the requirement that the labels must be equal; a directed edge expresses the dependency of the target's label on that of the source, that is, whatever type the source node is labeled with, the target node will be labeled in the same way. Whenever new type information is added to a type-flow graph by propagating labels along edges, the constraints represented by these edges have been satisfied so that the edges are not needed anymore. Similarly, after having assigned one new type variable to a group of nodes that are connected by undirected edges, these

[1]The directed edge indicates that the type for the result node is completely determined the result type of f's function type. The use of directed edges helps to simplify the type inference process since it generally reduces the number of required graph restructuring steps.

undirected edges have also served their purpose and are not needed anymore.[2] Consequently, we can remove these unneeded edges from the graph to simplify further operations by enhancing the graphs comprehensibility. In addition to unneeded edges, unconnected nodes (except the result node) can also be removed from the graph. For small examples like the one we are currently working with the effects are not that dramatic, but for larger graphs, edge and node removal can significantly simplify the inference process.

Returning to our example, we can observe that the edge connecting the nodes f_1 and g_2 and the edge between g_1 and x can be removed. We therefore obtain the type-flow graph shwon in Figure 3.[3]

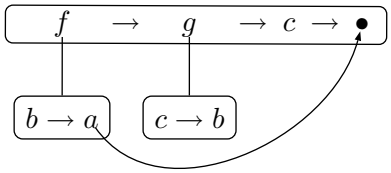

Figure 3: Type-flow graph after node labeling.

The next step is to propagate node labels via edges. We can propagate node labels via directed and undirected edges. Propagation works by labeling an unlabeled node with the label of its predecessor. In our example, we can first propagate a (the label of f_2) to the node r. Moreover, since the labels of the neighbors of f and g are saturated, we can also propagate the type $b \to a$ to f and the type $c \to b$ to g. After these three propagations, the remaining three edges are not needed anymore and can be dropped from the type-flow graph. The removal of the two undirected edges leaves the lower two nodes unconnected from the graph component that contains the result node, so that they can be removed, too. In general, propagation can lead to new saturated type expressions, enabling further propagations. However, in our example no unlabeled nodes are left, so no further propagation is possible, and we obtain the type-flow graph shown in Figure 4.

$$(b \to a) \to (c \to b) \to c \to a$$

Figure 4: Result type of composition combinator.

At this point, since the result node is labeled and since the graph contains no conflicts, the type-

[2]Another view is that assigning a fresh type variable to a group of nodes is actually an operation that consists of two consecutive steps, namely first assigning a variable to one node and then propagating it over the undirected edges to all other nodes in the group.

[3]Note that variable names, like f, g, and x, which we have kept in the type-flow graph for convenience, are *not* node labels but node identifiers. In this paper we use the type variables a, b, and c. All other lower case characters are node identifiers.

inference algorithm terminates successfully and reports the type of the function, which is obtained by the label of the result node of the lambda expression's type-flow graph.

Compared to the traditional type-inference algorithm, the described graph-labeling method is shorter and employs a more concise notation. Moreover, we do not have to maintain an explicit type environment throughout the algorithm. The type environment is represented rather implicitly through node labels. Unification requirements are represented by edges in the type-flow graph. Since these requirements are resolved in many cases through propagation, the use of unification is effectively reduced to a more moderate level.

Another observation is the following. We have shown the type inference in several small steps. For small examples like the one just considered, one would probably not redraw the whole graph in each step, but rather successively annotate just one graph. On the other hand, in more complex situations, producing intermediate results might be helpful. The point is that the described method supports any level of detail in the description of an inference, which is more difficult to achieve for the traditional algorithm.

One question remains: Why do we need directed as well as undirected edges? The answer is that directed edges indicate a possible type flow in only one direction, while an undirected edge, such as $\{v, w\}$, initially allows the flow of type information in either direction. Then later, during the execution of the algorithm, one of the nodes, say v, will be labeled. In that case, one of three things can happen. If w is unlabeled, v's node label can flow to w, that is, the label is copied. (This can happen also if v still contains unlabeled nodes in which case the undirected edge is moved and distributed into the type expression.) If w is labeled and the outermost type constructors of v's and w's label agree, the undirected edge $\{v, w\}$ is removed and new undirected edges are inserted between the corresponding arguments of v's and w's type (if any). Otherwise, we have found a type error. We illustrate these issues with more examples in later sections of this paper.

One important difference between the just presented algorithm and the traditional algorithm that explains much of their different style is the order in which the typing of subexpressions is performed. The traditional algorithm works in a syntax-directed way, which often forces to delay typing decisions and to resolve them later through unification. In contrast, the typing decisions in our algorithm are dependency-driven—the type-flow graph is essentially processed in topological order, which reduces notational and bookkeeping overhead to a minimum.

The rest of this paper is structured as follows. In Section 2 we discuss related work. In Section 3 we introduce type-flow graphs as a representation for type specifications of lambda terms. In Section 4 we introduce a type-inference algorithm that is based on a stepwise relabeling of the nodes of type-flow graphs. In Section 5 we extend the presented type-inference method to non-closed lambda terms by including predefined functions. We identify the notion of *normalized* type-flow graphs to avoid the re-checking of predefined functions. In Sections 6 and 7 we extend the method further to deal with recursive and polymorphic function definitions, respectively. Finally, in Section 8 we present some conclusions and comment on future work.

2 Related Work

Graphs have been used a lot as representations for lambda terms. Wadsworth [21] introduced a representation that is the basis for graph reduction. This representation allows the sharing of subexpressions. Lamping extended the representation by control nodes to avoid unnecessary copying [12]. These graphs are similar to the more general notion of interaction nets [11].

All these graph representations were used to study or improve the implementation of lambda term (graph) reduction; see also [1, 19]. A visual representation for lambda calculus (called VEX) was proposed in [4]. Through the use of nested circles this notation allows for an easy identification of scope and free and bound variables. On an abstract level the notation corresponds closely to other graph representations of lambda calculus and can be used to give a concise denotational semantics for lambda calculus [6].

The use of graphs in the context of type systems and type inference can be distinguished according to what information is being represented by graphs.

Most common is the use of graphs as a supporting data structure for type inference or for type explanation. Chitil [2] uses graphs to represent explanations of type inferences with the goal to produce better error messages. To this end he defines a version of the Hindley/Milner type system that is compositional. Flow graphs are used by Walz and Johnson to locate the most likely source of a type error [22]. A graph representation of types has been used in [10] to represent well-typed lambda terms by homomorphisms from graphs that represent lambda terms. In the work of McAdam [14, 13], graphs are used to represent lambda terms *and* types. The edges in the type graphs are labeled by terms that serve as explanations for the types. During type inference, the type graphs constantly grow, that is, there is no process of graph simplification, which makes perfectly sense for the described application of gathering information for type (error) explanation. However, for a description of the type-inference process itself, the used graph structures are too large and complex. Duggan and Bent [5] use graphs to represent the results of unifications, but graphs are not used to represent typing constraints. Again, the goal is to exploit the graph information to obtain explanations of type errors. The graph representation is derived from the one that is used in the almost linear time unification algorithm that was invented by Huet in his dissertation and is described in [9].

The use of graphs in this paper is to express type equality or type-flow constraints. Heeren et al. take a constraint-based approach [8] to improve type-error messages. They use a graph structure, called *equality graph*, to represent equality constraints between type variables and type constants. The graph concept we use in this paper differs in several respects: First, we represent constraints between arbitrary type expressions. To this end, we have developed a more sophisticated hierarchical graph structure that allows node labels to be expressions that may themselves contain nodes. Second, our graphs generally evolve during the type-inference process while their graphs are static. Finally, the graphs we are using are intended to be a direct communication device for users and not just an implementation structure. A similar use of graphs appears in [17] where constraint graphs are used to represents sets of typing constraints. However, these constraint graphs are also flat, static graphs. Solutions to the represented constraints are obtained through an automaton

that computes assignments of types to nodes. No graph operations are employed, and no graph simplification takes place. The most closely related graph representation appears in the Ph.D. thesis by Choppella who represents the type-unification problem by a *unification graph* [3]. Based on the work of Port [18], these unification graphs represent types. Like the graphs used in [8, 17], unification graphs are unstructured, that is, each vertex represents an unstructured type or a type constructor. In addition to edges expressing constraints between type nodes, the unification-graph representation contains also derived edges and paths. The graph is not simplified during the type-inference process. In contrast, unification is used to construct a closure relation of the represented equational constraints and thus grows more and more complex whereas in this paper the goal is to simplify graphs by repeatedly applying rewrite rules.

Finally, a use of graphs for explaining the process of type inference appears in the book by John Mitchell [16, Chapter 6] where he uses a graph notation for lambda expressions in an informal way to discuss the types of lambda terms.

3 Type-Flow Graphs

A *type-flow graph* is a graph to represent a specification of the type of a lambda expression. Nodes are labeled by type expressions in which type variables may be replaced by unlabeled nodes.

We are working with a set of type expressions T that is defined as the set of terms generated by an otherwise unspecified type signature Σ of type constructors $\sigma_1, \ldots, \sigma_n$ and a set of type variables A, that is, $T = T_\Sigma(A)$. The corresponding set of type expressions T_V to be used in type-flow graphs is obtained from T by allowing nodes in addition to type variables in terms. We assume that nodes are drawn from a set V, which is required to be disjoint from A. Therefore we define $T_V = T_\Sigma(A \cup V)$. For the type-inference algorithm we need a refinement operation that moves edges between nodes labeled by type expressions down to (nodes contained in) the subexpressions. The result of this operation is a sequence of updates to the current graph. In the following definition t ranges over types, and v and w represent nodes. Note that the unbound w on the right-hand side in the third case denotes a fresh node, that is, we require $w \notin V$.

$$\sigma(t_1, \ldots, t_n) \Downarrow \sigma'(t'_1, \ldots, t'_n) = \begin{cases} t_1 \Downarrow t'_1; \ldots; t_n \Downarrow t'_n & \text{if } \sigma = \sigma' \\ \textit{type error} & \text{otherwise} \end{cases}$$

$$v \Downarrow w = E \oplus \{v, w\}$$

$$v \Downarrow t = E \oplus \{v, w\}; V := V \cup \{w\}; L(w) := t$$

$$t \Downarrow v = v \Downarrow t$$

In pictures representing type-flow graphs we also use for convenience $\bullet$ to represent unlabeled nodes. The symbol $\bullet$ serves as a kind of "anonymous node identifier" that allows us to depict an unlabeled node in a picture without having to invent a unique identifier for it.

The information that is expressed by using the same variable name at different places in a type expression is represented in type-flow graphs by an undirected edge between the corresponding nodes. Note that with this representation, type-flow graphs are compositional under the assumption that type variables cannot be shared among different type expressions.

A *type-flow graph* is a triple $G = (V, L, E)$ where V is a set (the *node set* of G), $L : V \to T_V$ is a partial function that assigns node labels to nodes, and $E \subseteq V \times V$ represents the set of edges in the graph. We sometimes use the notation $\{v, w\}$ to denote an undirected edge, which is represented in the graph by the two directed edges (v, w) and (w, v). This shorthand allows us to use more succinct expressions, such as $E \cup \{\ldots, \{v, w\}, \ldots\}$, in place of $E \cup \{\ldots, (v, w), (w, v), \ldots\}$. Let $\mathit{nodes} : T_V \to 2^V$ be a function that computes the set of nodes contained in a type expression. We require the node labeling function L to be closed with respect to G, that is, $\forall v \in V : \mathit{nodes}(L(v)) \subseteq V$. With $E(v) = \{w \mid (v, w) \in E\}$ we can determine the set of successors of node v. Likewise, $E^{-1}(w) = \{v \mid (v, w) \in E\}$ yields the set of w's predecessors. We write $L(v) = \bot$ whenever $v \notin \mathit{dom}(L)$.

We use the following auxiliary notation for manipulating type-flow graphs. Let $G = (V, L, E)$. Adding a node v labeled by t to a graph G is defined as follows.

$$v\langle t\rangle \oplus G = (V \cup \{v\} \cup \mathit{nodes}(t), L \cup \{(v, t)\}, E)$$

We assume that $(\{v\} \cup \mathit{nodes}(t)) \cap V = \varnothing$. Correspondingly, we define $v \oplus G = (V \cup \{v\}, L, E)$ for adding an unlabeled node v to G. We use a similar notation to add directed and undirected edges to a graph.

$$(v, w) \oplus G = (V, L, E \cup \{(v, w)\})$$
$$\{v, w\} \oplus G = (V, L, E \cup \{\{v, w\}\})$$

The operation is well defined only if $\{v, w\} \subseteq V$.

To illustrate the graph-theoretic definition we show the type-flow graph of the composition combinator.

$$\begin{aligned}&(\{v, f, g, x, r, f', f_1, f_2, g', g_1, g_2\},\\ &\{(v, f \to g \to x \to r), (f', f_1 \to f_2), (g', g_1 \to g_2)\},\\ &\{\{f, f'\}, \{g, g'\}, \{f_1, g_2\}, \{g_1, x\}, (f_2, r)\})\end{aligned}$$

Next we describe the algorithm $\mathcal{F}$ to construct a type-flow graph from a lambda expression M. $\mathcal{F}$ computes a pair consisting of a graph representing the type specification for M and a node whose label represents the result type of M. This node is also called *result node* in the following.

The structure of type-flow graphs, their construction, and also the type-inference process is simplified by translating maximal groups of nested lambda abstractions and applications. We therefore assume in the following that patterns, such as $\lambda x_1 \ldots x_k.M$ or $M\ N_1 \ldots N_k$, always match a maximal number of parameters, respectively, arguments, that is, we assume that M is *not* a lambda abstraction in $\lambda x_1 \ldots x_k.M$ and M is not an application in $M\ N_1 \ldots N_k$. This assumption is not essential—the type inference algorithm works as well in the general case—but it makes type inferences generally simpler. Moreover, we assume that all lambda-bound variables in a lambda term are distinct. This property simplifies the translation into type-flow graphs and can easily be achieved by appropriately renaming variables. Note that renaming a lambda term M is not problematic in this context since the goal is to construct a graph representing M's type, which is not affected by the choice of variable names.

We provide two equivalent definitions for $\mathcal{F}$. First, we give a visual definition of the algorithm, which is helpful for two reasons: (i) it helps understanding the following abstract graph-theoretic

definition and (ii) it serves as a construction guide for drawing type-flow graphs when performing type inference "by hand". We employ the following visual notation.

- A rectangular, shaded box $\boxed{M}$ indicates the application of the transformation $\mathcal{F}$ to the enclosed lambda expression M.
- A black diamond ♦ on the border of a shaded box refers to the result node of the graph that is obtained by that translation.
- A rounded box $\boxed{t}$ represents a node that is labeled by a type expression t; unlabeled nodes are represented by • or by lowercase variables, such as v or w.
- Arrows represent directed edges to be created, whereas lines represent undirected edges.
- A black diamond ♦ next to a node marks the node to represent the result type of the constructed type-flow graph.

The translation of an abstraction $\lambda x_1 \ldots x_k.M$ creates one node v that is labeled by a function type corresponding to the number of parameters of the abstraction ($k+1$ defines the number of unlabeled nodes in the generated type expression) and a graph G_M that is obtained by translating the body of the abstraction. Since the result type of a function is given by the type of the function's body, a directed edge is added from the result node of G_M to the last node in v's label. The result type of the abstraction is given by the label of v. Therefore, v is marked as the result node, see Figure 5.

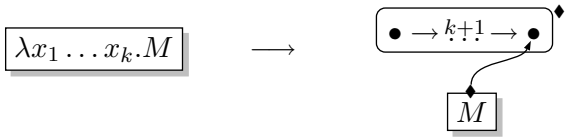

Figure 5: Translation of abstractions.

An application $M\ N_1 \ldots N_k$ is translated into graphs for the applied function M and all k arguments $N_1, \ldots, N_k$. An application is well typed if the type of M is a function type of (at least) k arguments and if the types of the arguments match the corresponding parameter types. The first requirement is established in the generated type-flow graph by creating a node v that is labeled with a function type and by adding an undirected edge between v and the result node of M. The second constraint is realized by adding an undirected edge between the result node of each argument graph and the corresponding parameter node in v's label, see Figure 6. The result type of the whole application is given by the result type of M, which is reflected by marking the last unlabeled node of v's type expression as a result node.

Finally, the translation of a variable shown in Figure 7 simply causes the unlabeled node that represents the corresponding parameter to be marked as a result node. This marking will then be

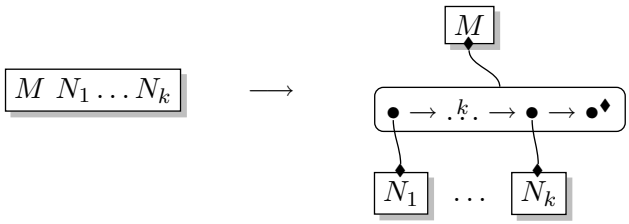

Figure 6: Translation of applications.

exploited by the algorithm to attach an edge to that node. Since we assume that all lambda-bound variables have unique names, this node is uniquely determined. Note that this step may fail in the case of unbound variables. In that case $\mathcal{F}$ returns an error.

Figure 7: Translation of variables.

The translation can also be generalized to work with terms that contain the same variable multiple times by maintaining an stack of node/variable pairs. However, this generalization requires additional technical machinery and does not help otherwise.

Next we give the formal definition of the type-flow graph construction. $\mathcal{F}$ takes as input an already constructed type-flow graph and the lambda expression to be added to that graph. The type-flow graph is constructed inductively by scrutinizing the lambda expression and applying one of the following rules. Unbound variables in the definitions, such as $w, v, v_1, \ldots, v_{k+1}$, denote fresh nodes that are not contained in any of the involved graphs yet.

The type of a lambda abstraction is a function type whose result type is given by the type of the lambda body. Therefore the following construction creates for a lambda expression of k parameters a new node v labeled by a corresponding function type, that is, $x_1 \to \ldots \to x_k \to w$, and connects the node r that represents the result type of the lambda body M with the result type node w by a directed type-flow edge.

$$\begin{aligned}&\mathcal{F}(G, \lambda x_1 \ldots x_k.M) = ((r, w) \oplus G', w)\\ &\quad \text{where } (G', r) = \mathcal{F}(v\langle x_1 \to \ldots \to x_k \to w\rangle \oplus G, M)\end{aligned}$$

The type-flow graph for an application generates graphs for all k arguments and links each node r_i, representing the type of the argument term N_i, to the corresponding node in the function type $v_1 \to \ldots \to v_{k+1}$, which is used as a label of the node v, which is to be linked to the node r

representing the result type of M.

$$\begin{aligned}\mathcal{F}(G, M\ N_1 \ldots N_k) &= (\{r_0, v_0\} \oplus \ldots \oplus \{r_k, v_k\} \oplus v_0\langle v_1 \to \ldots \to v_{k+1}\rangle \oplus G_k, v_{k+1}) \\ \text{where } (G_0, r_0) &= \mathcal{F}(G, M) \\ (G_i, r_i) &= \mathcal{F}(G_{i-1}, N_i) \text{ for } 1 \le i \le k\end{aligned}$$

Finally, the type-flow graph of a variable does not change the graph, but simply returns the corresponding node as a result node if it is present in the graph. With $G = (V, L, E)$ we define:

$$\mathcal{F}(G, x) \ = \ \begin{cases}(G, x) & \text{if } x \in V \\ \mathit{error} & \text{otherwise}\end{cases}$$

4 Type-Inference with Type-Flow Graphs

The type-inference algorithm works by successively labeling nodes in the type-flow graph for a lambda expression. The basic idea is to repeatedly label unlabeled nodes that do not depend on the type of any other node by new type variables and to propagate type expressions via edges to other unlabeled nodes.

In the description of the algorithm we employ the following characterizations of nodes in a type-flow graph. For each unlabeled node v, the set of its *equivalent nodes* is defined as the set of all unlabeled nodes that can be reached from v via path of undirected edges, that is, with $v \equiv w \ :\Longleftrightarrow\ \{v, w\} \in E$ we have

$$\mathit{equiv}(v) = \{w \in V \mid L(w) = \bot \wedge v \equiv^* w\}$$

where $\equiv^*$ denotes the reflexive and transitive closure of $\equiv$.

An node is said to be *unconstrained* if it does not have a labeled predecessor.

$$\mathit{uc}(v) = E^{-1}(v) \cap \mathit{dom}(L) = \varnothing$$

Note that the use of E^{-1} (instead of E) ensures that possible directed incoming edges are caught. (Outgoing directed edges do not constrain a node.)

A node is called a *free node* if all of its equivalent nodes are unconstrained and if they are part of node labels of unconstrained nodes only. The formal definition is as follows.

$$\mathit{free}(v) = \forall w \in \mathit{equiv}(v) : \big(\mathit{uc}(w) \wedge \forall u \in V : w \in \mathit{nodes}(L(u)) \implies \mathit{uc}(u)\big)$$

If an unlabeled node is free, it can be labeled by a fresh type variable.

Finally, a *saturated node* is a node whose label does not contain any unlabeled nodes, that is,

$$\mathit{sat}(v) = \{v\} \cup \mathit{nodes}(L(v)) \subseteq \mathit{dom}(L)$$

In the propagation step, which promotes node labels, saturated nodes are preferred to simplify and to accelerate the type-inference process.

ALGORITHM $\mathcal{G}$ (Graph-Based Type Inference)
INPUT. A lambda expression M.
OUTPUT. The type for M or *type error*.
METHOD.
First, compute the type-flow graph G for M. Let r be the result node of G. Repeatedly apply the first applicable of the following four steps. After each step remove single, isolated nodes (except r).

1. *Recognize type error.* If an edge connects two incompatible nodes, report a type error and stop. Two nodes are incompatible if (a) they are labeled with type expressions that do not match or (b) one node is not part of the label of the other.
2. *Label.* Label a group of free nodes with a new type variable, that is, select $\mathit{equiv}(v) \subseteq V$ with $\mathit{free}(v)$ and assign for all $w \in \mathit{equiv}(v) : L(w) := a$ for an $a \notin L(V)$. Let $E := E - \{\{u, w\}\}$ for all $u, w \in \mathit{equiv}(v)$.
3. *Propagate.* Label an unlabeled node with the type expression of one of its (preferably saturated) predecessors, that is, select $u \in E^{-1}(v)$ with $L(v) = \bullet$ and let $L(v) := L(u)$. Let $E := E - \{(u, v)\}$. Favor the propagation of saturated predecessors.
4. *Refine.* Distribute undirected edges to nodes in type expressions, that is, select $\{v, w\} \in E$ with $L(v) \neq \bot$ and $L(w) \neq \bot$, let $E := E - \{\{v, w\}\}$, and perform the operations determined by $v \Downarrow w$. Note that for $n = 0$ this step amounts to just checking constraints on saturated type expressions, which is reflected in the first case of the definition of $\Downarrow$.

If $L(r)$ is saturated, return $L(r)$. Otherwise, report a type error.

Figure 8: Type Inference Algorithm $\mathcal{G}$.

The graph-based type-inference algorithm $\mathcal{G}$ shown in Figure 8 essentially consists of repeatedly applying one of two node-labeling steps: (i) labeling free nodes with new type variables or (ii) propagating node labels along type-flow edges. In some situations, the graph structure is refined to enable further relabeling or the identification of type errors.

We have already illustrated in Section 1 how $\mathcal{G}$ can successfully infer types. Therefore, we now present an example that demonstrates the identification of a type error. Consider the lambda term $\lambda x.x\ x$, which is ill typed. To apply $\mathcal{G}$, we first have to create the type-flow graph. The process is illustrated in Figure 9.

For the further discussion we use, in addition to x, the names r, y, and z (top to bottom, left to right) for the unlabeled nodes in the resulting graph and f for the labeled node at the bottom.

Now we can try to apply the different steps of the algorithm $\mathcal{G}$. We cannot yet recognize a type error because in order to identify condition (a) we have to find an edge that connects two labeled nodes, which does not exist in the above type-flow graph. To apply the second step, we find that z is free because (i) it does not have any labeled predecessor and is thus is itself unconstrained and (ii)

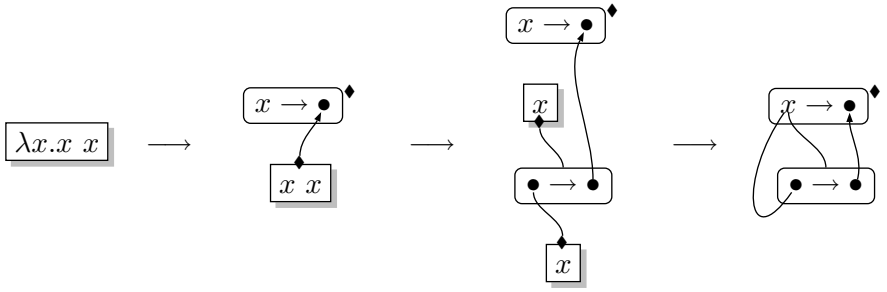

Figure 9: Application of $\mathcal{F}$.

although it is part of f's node label $y \to z$, the node f is unconstrained, too, because it only has an unlabeled node, namely x, as a predecessor. Therefore, we can label z with a. Since $\mathit{equiv}(z) = \{z\}$, E does not have to be adjusted. Note that y is not free and cannot be labeled because one of its equivalent nodes, namely x, is constrained (since it is connected by an edge to the labeled node f). Therefore, we obtain the type-flow graph shown in Figure 10.

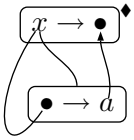

Figure 10: Type-flow graph after labeling.

In the next round of the algorithm, the first step that applies is propagation. Since we prefer the propagation of saturated nodes, we select z and propagate the label a to r, which produces, after removing the edge (a, r), the type-flow graph shown in Figure 11.

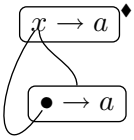

Figure 11: Type-flow graph after propagation.

In the next iteration, propagation is again the first possible step. This time we propagate the label $y \to z$ of the unsaturated node f to x. Note that for convenience we may "inline" labeled nodes in type expressions. For example, instead of talking about "f with $L(f) = y \to z$ and $L(z) = a$", we can more succinctly as well refer to "f with $L(f) = y \to a$". Formally, we can define the following equivalence relationship on type-flow graphs. For $\sigma \in \Sigma, t_1, \ldots, t_n \in T_V, v \in V$ we define that the

following two type expressions are equivalent in a graph G:

$$\sigma(t_1, \ldots, t_i, \ldots, t_n) \cong \sigma(t_1, \ldots, v, \ldots, t_n)$$

iff $L(v) = t_i$ and $v \notin \mathit{nodes}(\sigma(t_1, \ldots, t_i, \ldots, t_n))$ and if $E(v) = E^{-1}(v) = \varnothing$.

Returning to the example, after the propagation we obtain the type-flow graph shown in Figure 12. Note that after removing the edge $\{x, f\}$, f is isolated and can be removed from the graph. Note also that we cannot substitute in this case x by $L(x) = y \to a$ since $E(x) = \{y\} \neq \varnothing$.

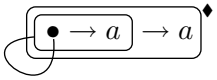

Figure 12: Type-flow graph revealing the type error in $\lambda x.x\ x$.

In the last iteration, the first step applies since part (b) of the condition is satisfied, that is, we recognize the nodes y and x as being incompatible since they are connected by an edge while at the same time y is contained in $L(x)$. Therefore the algorithm stops at this point and reports a type error.

The correctness of the algorithm follows from the following theorem that relates typeability in the Hindley/Milner type system to the computation of a saturated result node by $\mathcal{G}$. For completeness the type inference rules of the Hindley/Milner type system for application, abstraction, and variables are shown in Figure 13. These rules define a typing judgment $\Gamma \vdash M : t$ that expresses that M has type t under the typing assumptions in Γ where Γ is a list of variable/type pairs that realizes a mapping from variables into types.

$$\textsc{Var}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \textsc{App}\ \frac{\Gamma \vdash M : t' \to t \quad \Gamma \vdash N : t'}{\Gamma \vdash M\ N : t} \qquad \textsc{Abs}\ \frac{\Gamma, x : t' \vdash M : t}{\Gamma \vdash \lambda x.M : t' \to t}$$

Figure 13: Hindley/Milner typing rules.

In the following we write G_Γ for the graph $(\mathit{dom}(\Gamma), \Gamma, \varnothing)$, which consists of a labeled node for each variable in Γ.

Theorem 1 $\Gamma \vdash M : t \iff L(r) = t$ *where* $(V, L, E) = \mathcal{G}(G)$ *and* $(G, r) = \mathcal{F}(G_\Gamma, M)$.

Proof. The proof is by induction over the structure of lambda terms.

For $M = x$ the premise of the typing rule VAR requires $\Gamma(x) = t$. We observe that $\mathcal{F}(G_\Gamma, x) = (G_\Gamma, x)$ will just mark x as a result node. Since in G_Γ we have $L = \Gamma$, it follows that $L(r) = L(x) = \Gamma(x) = t$, which proves the "$\Longrightarrow$" direction of this case. On the other hand, since $\mathcal{F}(G_\Gamma, x)$ does not change the graph G_Γ, the fact that $L(r) = t$, which is equivalent to $\Gamma(x) = t$ in this case, provides the premise for rule VAR to conclude $\Gamma \vdash x : t$, which proves the "$\Longleftarrow$" direction of this case.

If $M = N\ N'$, we know by typing rule App that $\Gamma \vdash N : t' \to t$ and $\Gamma \vdash N' : t'$. By induction we can assume that $L(r) = t' \to t$ and $L'(r') = t'$ where $(V, L, E) = \mathcal{G}(G)$ with $(G, r) = \mathcal{F}(G_\Gamma, N)$ and $(V', L', E') = \mathcal{G}(G')$ with $(G', r') = \mathcal{F}(G_\Gamma, N')$.

$\mathcal{F}$ will construct the following type-flow graph for M to which we can first apply the induction hypothesis (1). The resulting graph can then be reduced through (2) propagation, (3) refinement, (4) the special case of refinement that amounts to constraint checking for the nodes labeled with t', and finally (5) propagation of t to v_2. This process is illustrated in Figure 14.

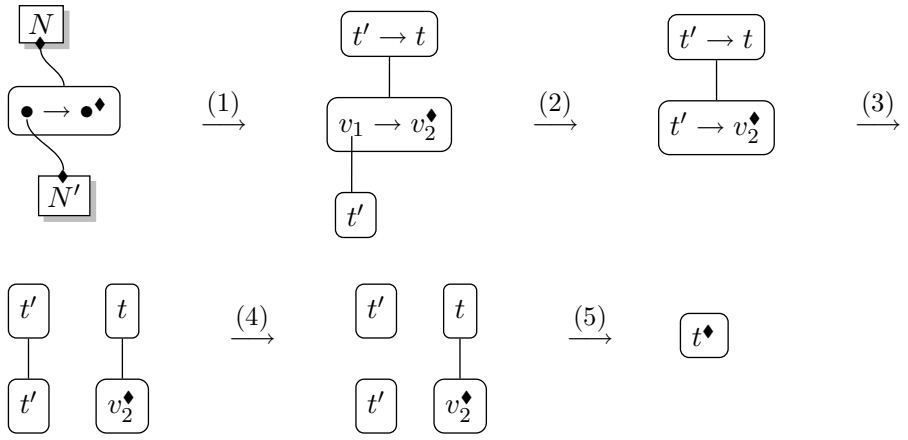

Figure 14: How $\mathcal{G}$ works on applications.

We can observe that the label of the result node is t, which proves the "$\Longrightarrow$" direction of this case. To prove the "$\Longleftarrow$" direction of this case, we note that $\mathcal{F}$ will start in the same way as illustrated above. Then, in order for $\mathcal{G}$ to be able to label the result node with t, the type of N must be a function type, say $t' \to t''$, because otherwise refinement would not be possible. Moreover, the result type t'' must be t so that propagation can move t to the result node. Similarly, the argument type t' must be equal to the type inferred for N' so that refinement will be successful for both function types' argument types. This line of reasoning leads to the type-flow graph shown above after "(1)". The labels for the result nodes of the graphs for N and N' establish by using the induction hypothesis the premises for the App typing rule, which can then be applied and leads to the conclusion $\Gamma \vdash N\ N' : t$ to prove this case.

If $M = \lambda x.N$, we know by typing rule Abs that $\Gamma, x : t' \vdash N : t$. By induction we can assume that $L(r) = t$ where $(V, L, E) = \mathcal{G}(G)$ with $(G, r) = \mathcal{F}(G_{\Gamma, x:t'}, N)$.

$\mathcal{F}$ builds the type-flow graph for M that is shown in Figure 15. To this graph we can apply the induction hypothesis (1). The resulting graph can then be reduced through propagation (2).

We can observe that the label of the result node is $t' \to t$, which proves the "$\Longrightarrow$" direction of this case. To prove the "$\Longleftarrow$" direction, we observe that $\mathcal{F}$ builds a type-flow graph as shown above. Then $\mathcal{G}$ will determine some type t for N, which is propagated to the result node. In doing so, sooner or later some type t' will be assigned to the node x (for example, through propagation

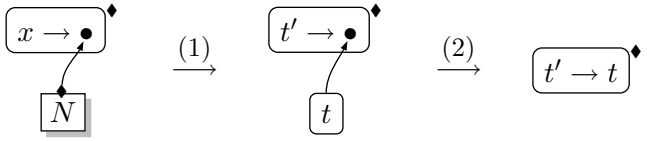

Figure 15: How $\mathcal{G}$ works on abstractions.

or through labeling x should it be free). The fact that the result node of the type-flow graph for N is labeled t while x is labeled t' establishes by using the induction hypothesis the premise for the ABS typing rule, which can then be applied and leads to the conclusion $\Gamma \vdash \lambda x.N : t' \to t$ to prove this case and the theorem. □

5 Using Predefined Functions

So far we have considered the type inference for closed lambda terms. In this section we demonstrate how the type-flow-graph approach can be extended to infer types of expressions that contain predefined symbols. The basic idea is very simple: Create type-flow graphs for all predefined symbols to be used and insert them during the inference process.

However, in this simple and naive form the approach is not practical since using the type-flow graphs as generated by the algorithm $\mathcal{F}$ described in Section 3 would require to re-typecheck the predefined operations on every use, which is not desirable. Fortunately, we can use a normalized form of type-flow graphs that represent the type information in solved form. A normalized type-flow graph can be constructed from a type by the algorithm that is shown in Figure 16.

ALGORITHM $\mathcal{N}$ (Type-Flow-Graph Normalization)
INPUT. A type expression t.
OUTPUT. A normalized type-flow graph G for t.
METHOD.
(1) Replace each type variable in t by a new node. Let V be the set of inserted nodes, and let $t' \in T_V$ the changed type expression.
(2) Insert undirected edges so that all nodes in V that have replaced the same type variable are connected. Let E be the set of all generated edges.
(3) Return $(V \cup \{v\}, \{(v, t')\}, E)$ where $v \notin V$.

Figure 16: Generating normalized type-flow graphs.

As an example consider the normalized type-flow graph that is obtained for the function composition combinator, see Figure 17.

Normalized type-flow graphs for constants like numbers or booleans are given by labeled nodes that will be shown in type-flow graphs just by their label. As further examples we show the normalized type-flow graphs for a conditional and for a pair constructor in Figure 18.

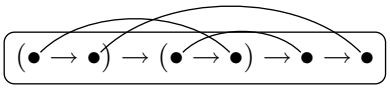

Figure 17: Normalized type-flow graph for composition combinator.

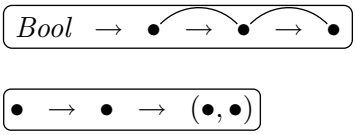

Figure 18: Normalized type-flow graphs for conditional and pair.

As an example application of $\mathcal{G}$ consider the task of inferring the type of

$$\mathit{compose\ not\ even}$$

where $\mathit{compose} = \lambda f.\lambda g.\lambda x.f\ (g\ x)$ is the composition combinator. The type-flow graph shown in Figure 19 is obtained from the translation of the application by inserting the normalized type-flow graphs for *compose*, *not*, and *even*.

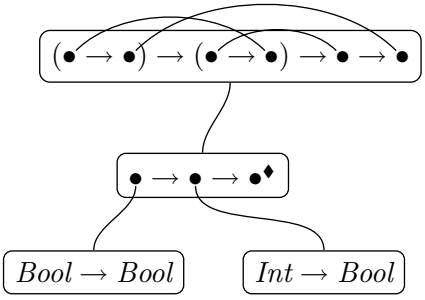

Figure 19: Application of composition.

The first applicable step of $\mathcal{G}$ is to propagate the types of *not* and *even*, which yields the type-flow graph shown in Figure 20 after removing two undirected edges and the unconnected nodes.

Why is it not possible to label the groups of equivalent nodes in the topmost node with fresh type variables? Because they are not free: The second condition of freeness is violated, that is, the nodes are contained in a label of a node that is connected to a labeled node.

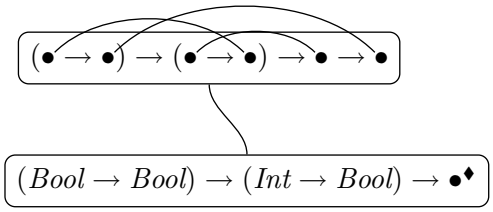

Figure 20: Type-flow graph after two propagations.

Next we can repeatedly refine over the function type constructors, which causes a new node with label $\bullet \to \bullet$ to be introduced in the upper node (following the third case in the definition of the operation $\Downarrow$), see Figure 21.

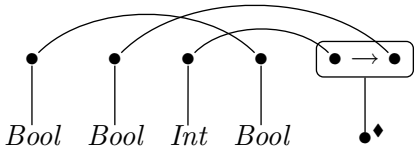

Figure 21: Type-flow graph after refining steps.

Now we can propagate the constant types and obtain the graph shown in Figure 22.

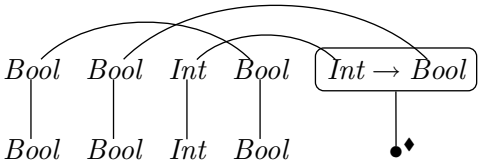

Figure 22: Type-flow graph after propagation of constant types.

The next steps are refinements in the special form of constraint checking. After that a propagation of $Int \to Bool$ to the result node, followed by edge and isolated node removal, leads to the final graph shown in Figure 23.

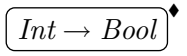

Figure 23: Result node.

6 Recursion

Recursion means for the process of type inference to make the type node for the function being defined accessible in the inference of the function's definition. In this section we consider only monomorphic definitions and focus on the treatment of recursion.

Consider a let expression **let** $x = d$ **in** e. In the flow-type graph we will have a node v_d that is labeled by a type expression t_d. All references to x in d can be treated in the type-flow graph by assuming x is a predefined symbol, which would mean to copy the corresponding type-flow graph. However, instead of copying we can use the instance that is already present in the current type-flow graph, namely v_d, and connect any edges to v_d.

As an example we consider the type inference for the function *length*, which is defined as follows.

$$length = \lambda xs.\textbf{if } null\ xs \textbf{ then } 0 \textbf{ else } succ\ (length\ (tail\ xs))$$

First, we construct the type-flow graph for this function definition, see Figure 24.

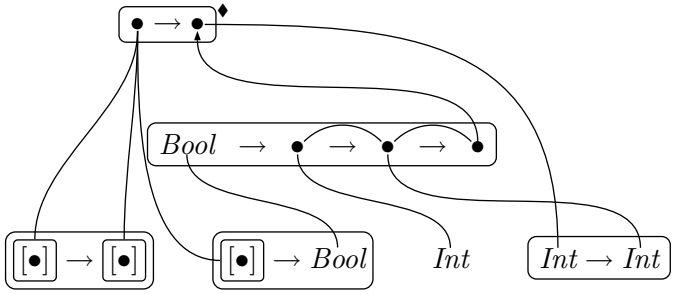

Figure 24: Type-flow graph for the *length* function.

Note how the type specification for the **else** branch of the definition appears somehow "distributed" in the translation: First, the recursive call to *length* is represented by the function-type node for *tail* whose argument and result types are both connected to the node representing the type of *xs* (since *tail* is applied to *xs* and since its result is used by *length*). Second, the application of *succ* to (*length* (*tail xs*)) is represented by an edge between the argument type of the $Int \to Int$ labeled node representing *succ* to the result node of *length*'s type. In particular, it is striking that different occurrences of one symbol are represented by one node with additional edges reflecting the constraints of all uses, which shows that type-flow graphs do not reflect the exact structure of expressions, but only their typing relationships.

Let us now apply the algorithm $\mathcal{G}$. First, we propagate the type *Int* to unlabeled nodes. The edges over which *Int* was propagated are deleted after propagation, as well as the now disconnected node *Int* itself, see Figure 25.

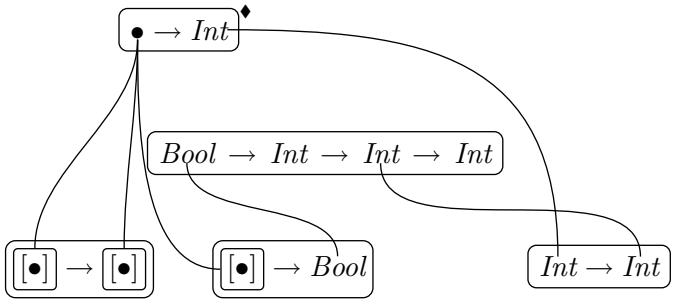

Figure 25: Type-flow graph after propagation and edge/node removal.

Next we can refine at three places. In fact, in all cases the refine step applies to type constructors without argument, which means to just check constraints. The corresponding edges can be removed, which leaves the type nodes for the conditional and for *succ* isolated, so that they can be removed from the graph, too. We are left with the type-flow graph that is shown in Figure 26.

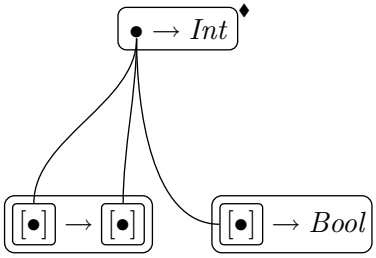

Figure 26: Type-flow graph after constraint checking.

At this point, propagation applies to all remaining edges, which means to label the unlabeled node by [•]. After that, refinement over the list type constructor moves the three undirected edges inside the type expression, which yields the type-flow graph shown in Figure 27.

Since all unlabeled nodes are free, the second step of $\mathcal{G}$ applies, which allows us to label them with a fresh type variable a. Removing the edges that connect all relabeled nodes leaves the type nodes for *tail* and *null* unconnected so that they can be removed. Therefore, we are finally left with the result node that is labeled with the resulting type expression as shown in Figure 28.

We observe that in order to deal with recursion, no changes were required to the algorithm $\mathcal{G}$, and only a minimal change was needed for the algorithm $\mathcal{F}$, namely linking recursive calls by edges to the corresponding node in the graph. A corresponding extension to the definition of $\mathcal{F}$ to deal

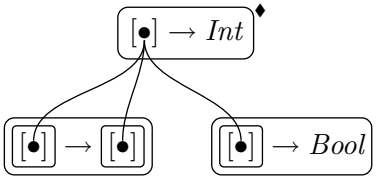

Figure 27: Type-flow graph after refinement.

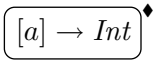

Figure 28: Result node carrying the type of the *length* function.

with **let** expressions will be presented in the next section.

7 Polymorphism

Parametric polymorphism in the Hindley/Milner type system means the ability to instantiate generic type variables at different places to different types. In the algorithm $\mathcal{G}$ we can realize this behavior by creating different copies of the type node for the polymorphic function. But how can we ensure that only type variables are generalized that are "not free in the type environment Γ", speaking in terms of the traditional algorithm? The somewhat surprising answer is that this happens automatically! The reason is that free type variables are represented by unlabeled nodes that are not copied, so that different copies of type nodes for the let-bound variables will eventually be linked to one common node that would lead to a conflict during propagation.

First, we extend the definition of $\mathcal{F}$ to **let** expressions. In order to distinguish let-bound from lambda-bound variables, we extend the type-flow graph representation by an additional set C (as a fourth component), which will consist of all nodes that represent let-bound variables. The set C is initialized to $\varnothing$ and is not affected by other cases than the following.

$$\begin{aligned} \mathcal{F}(G, \textbf{let } x = M \textbf{ in } N) &= \mathcal{F}(G', N) \\ \text{where } ((V, L, E, C), r) &= \mathcal{F}(G, M) \\ G' &= (V, L, E, C \cup \{r\}) \end{aligned}$$

In addition, we have to extend $\mathcal{F}$'s definition for translating references to let-bound variables. In contrast to lambda-bound variables, which are just marked as a result node to prepare the addition of edges, nodes for let-bound variables are copied to enable the independent instantiations of different references to such variables. With $G = (V, L, E, C)$ we obtain the following definition. Again the

unbound variable v represents a fresh node.

$$\mathcal{F}((V,L,E,C),x) \;=\; \begin{cases} ((V \cup \{v\}, L \cup \{(v, L(x))\}, E, C), v) & \text{if } x \in C \\ ((V,L,E,C),x) & \text{if } x \in V - C \\ \mathit{error} & \text{otherwise} \end{cases}$$

As an example let us first consider how type inference works for the following expression.

$$\textbf{let } f = \lambda x.x \textbf{ in } (f\ 3, f\ \mathit{True})$$

In the type-flow graph for this expression shown in Figure 29 the type node for f is duplicated. Note also that the result node of the flow graph is not given by the node for the let body (the normalized type-flow graph for the pair constructor), but rather by the node that carries the result type of the pair constructor.

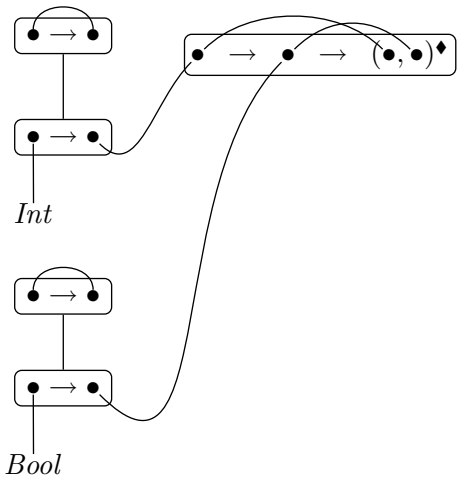

Figure 29: Type-flow graph for **let** expression.

The application of the algorithm $\mathcal{G}$ solves the typing problem in three steps by propagating both constant types, followed by a refinement, and finally another propagation, so that we eventually obtain the result shown in Figure 30.

$$\boxed{\mathit{Int} \;\to\; \mathit{Bool} \;\to\; (\mathit{Int}, \mathit{Bool})^{\blacklozenge}}$$

Figure 30: Final type-flow graph for **let** expression.

The result type of the let expression is given by the result node of the flow graph, which carries the label $(\mathit{Int}, \mathit{Bool})$.

To illustrate the proper treatment of free type variables we consider as another example a variation of the previous expression that is not typeable.

$$\lambda g.\textbf{let } f = g \textbf{ in } (f\ 3, f\ \mathit{True})$$

Since the type of g is not polymorphic, the type variable in f's type cannot be generalized, which prohibits the instantiation to different types in the pair.

In the type-flow graph for the above expression, we create a new copy for each reference to f in the pair constructor, but the referenced node that represents the type of g is not copied so that both type nodes for f are connected to the same node g, see Figure 31.

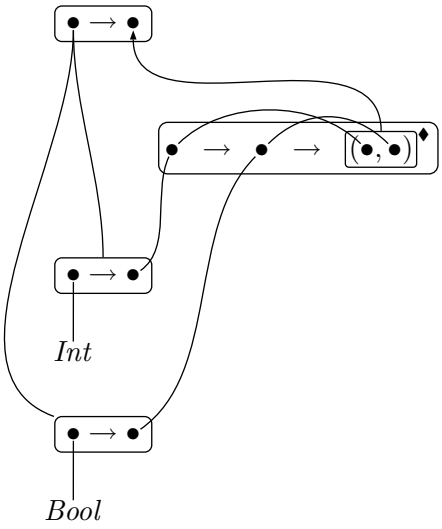

Figure 31: Type-flow graph for erroneous **let** expression.

After propagating *Int* and *Bool* and one of the resulting function types to the node representing g, refining we obtain the type-flow graph shown in Figure 32.

In a following refinement step between $\mathit{Int} \to \bullet$ and $\mathit{Bool} \to \bullet$ an edge will created between an *Int* and a *Bool* node, which causes $\mathcal{G}$ to report a type error. These examples show that the node-copying approach for polymorphic types works well, in particular, it respects the different treatment of generic and non-generic type variables.

8 Conclusions and Future Work

The presented type-inference algorithm $\mathcal{G}$ and the underlying structure of type-flow graphs is intended to serve the explanation of type inference in the classroom as well as in editing and reasoning tools for types. We believe that demand-driven processing of typing constraints and the iterative process are easier to understand and to perform (for humans) than the recursive, syntax-directed

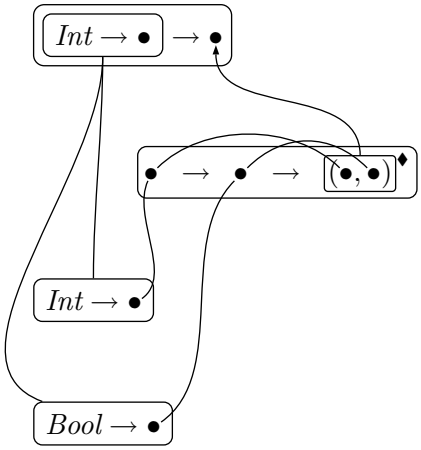

Figure 32: Intermediate type-flow graph revealing a type error.

approach of the traditional algorithm. Although the generation of type-flow graphs happens syntax-directed as well, the solving of the typing constraints is separated from this process, which simplifies the overall procedure.

We believe that the type-inference approach has potential for improvements. One direction for further investigations are sound, composite graph operations that can simplify type-flow graphs even faster. For example, it is correct to join both pairs of connected function nodes in the first let example from Section 7, which would simplify the graph immediately as shown in Figure 33.

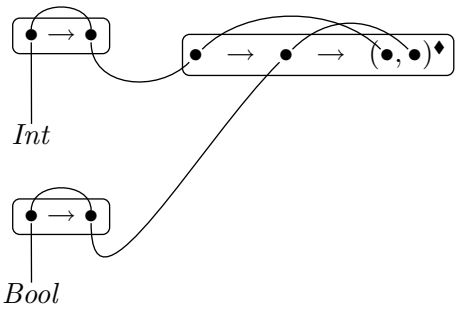

Figure 33: Shortcut type-flow graph.

The final result is now obtained by just propagating *Int* and *Bool*.

The presented approach seems to be well-suited for tool support. For example, specialized graph editors for creating type proofs or for animating type proofs can be developed. Besides the general graph-layout problem, a particularly important aspect is to provide means for zooming in and out of complex type proofs, which requires an appropriate concept of a hierarchy of type-flow

graphs to (re)present type inferences on different levels of abstractions. In this context we could also investigate whether the graph representation provides opportunities to improve type error messages.

Finally, one of our original motivations for investigating a graph-based type representation was to obtain more concise representations of type information to be used in the area of type-safe metaprogramming [7]. A simpler representation of typing derivations can be exploited to obtain an expressive, but still easily comprehensible notation for describing type changes and type changing operations.

References

[1] Z. M. Ariola and M. Felleisen. The Call-by-Need Lambda Calculus. *Journal of Functional Programming*, 7(3):265–301, 1996.

[2] O. Chitil. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *6th ACM Int. Conf. on Functional Programming*, pages 193–204, 2001.

[3] V. Choppella. *Unification Source Tracking With Application to Diagnosis of Type Inference*. Ph.D. Thesis, Indiana University, 2002.

[4] W. Citrin, R. Hall, and B. Zorn. Programming with Visual Expressions. In *11th IEEE Symp. on Visual Languages*, pages 294–301, 1995.

[5] D. Duggan and F. Bent. Explaining Type Inference. *Science of Computer Programming*, 27:37–83, 1996.

[6] M. Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing*, 9(5):461–483, 1998.

[7] M. Erwig and D. Ren. Type-Safe Update Programming. In *12th European Symp. on Programming*, LNCS 2618, pages 269–283, 2003.

[8] B. Heeren, J. Jeuring, D. Swierstra, and P. A. Alcocer. Improving Type-Error Messages in Functional Languages. Technical Report UU-CS-2002-009, Institute of Information and Computing Science, University Utrecht, Netherlands, 2002. Technical Report.

[9] K. Knight. Unification: A Multidsiciplinary Survey. *ACM Computing Surveys*, 21:93–124, 1989.

[10] W. Kahl. The Term Graph Programming System HOPS. In *Tool Support for System Specification, Development, and Verification*, pages 136–149, 1999.

[11] Y. Lafont. Interaction Nets. In *17th ACM Symp. on Principles of Programming Languages*, pages 95–108, 1990.

[12] J. Lamping. An Algorithm for Optimal Lambda Calculus Reduction. In *17th ACM Symp. on Principles of Programming Languages*, pages 16–30, 1990.

[13] B. J. McAdam. On the Unification of Substitutions in Type Inference. In *Int. Workshop on Implementation of Functional Languages*, LNCS 1595, pages 139–154, 1998.

[14] B. J. McAdam. Graphs for Recording Type Information. Technical Report ECS-LFCS-99-415, University of Edinburgh, 1999.

[15] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:248–375, 1978.

[16] J. C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, Cambridge, UK, 2003.

[17] J. Palsberg, M. Wand, and P. O'Keefe. Type Inference with Non-Strictural Subtyping. Report BRICS RS-95-33, BRICS, Department of Computer Science, University of Aarhus, 1995.

[18] G. S. Port. A Simple Approach to Finding the Cause of Non-Unifiability. In *5th Int. Conf. and Symp. on Logic Programming*, pages 651–665, 1988.

[19] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen. *Term Graph Rewriting – Theory and Practice*. John Wiley & Sons, Chichester, 1993.

[20] S. Thompson. *Haskell – The Craft of Functional Programming (2nd ed.)*. Addison-Wesley, Harlow, England, 1999.

[21] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. D.Phil. thesis, Oxford University, 1971.

[22] J. A. Walz and G. F. Johnson. A Maximum-Flow Approach to Anomaly Isolation in Unification-Based Incremental Type Inferenc. In *ACM Symp. on Principles of Programming Languages*, pages 44–57, 1986.

Appendix

The following equations illustrate an example application of the type-inference algorithm $\mathcal{W}$. $\mathcal{U}$ denotes the unification algorithm.

$$
\begin{aligned}
\mathcal{W}(\varnothing, \lambda f.\lambda g.\lambda x.f\ (g\ x)) &= (U, Ua \to t) \\
&\quad \text{where} \quad (U,t) = \mathcal{W}(\{f \mapsto a\}, \lambda g.\lambda x.f\ (g\ x)) \\
&\qquad\qquad = ([e \to d/a, c \to e/b], (c \to e) \to (c \to d)) \\
&= ([e \to d/a, c \to e/b], (e \to d) \to (c \to e) \to (c \to d)) \\
\mathcal{W}(\{f \mapsto a\}, \lambda g.\lambda x.f\ (g\ x)) &= (V, Vb \to t') \\
&\quad \text{where} \quad (V,t') = \mathcal{W}(\{f \mapsto a, g \mapsto b\}, \lambda x.f\ (g\ x)) \\
&\qquad\qquad = ([e \to d/a, c \to e/b], c \to d) \\
&= ([e \to d/a, c \to e/b], (c \to e) \to (c \to d)) \\
\mathcal{W}(\{f \mapsto a, g \mapsto b\}, \lambda x.f\ (g\ x)) &= (W, Wc \to t'') \\
&\quad \text{where} \quad (W,t'') = \mathcal{W}(\{f \mapsto a, g \mapsto b, x \mapsto c\}, f\ (g\ x)) \\
&\qquad\qquad = ([e \to d/a, c \to e/b], d) \\
&= ([e \to d/a, c \to e/b], c \to d) \\
\mathcal{W}(\{f \mapsto a, g \mapsto b, x \mapsto c\}, f\ (g\ x)) &= (TSR, Td) = ([e \to d/a, c \to e/b], d) \\
&\quad \text{where} \quad (R,s) = \mathcal{W}(\{f \mapsto a, g \mapsto b, x \mapsto c\}, f) = (\varnothing, a) \\
&\qquad\qquad (S,s') = \mathcal{W}(R\{f \mapsto a, g \mapsto b, x \mapsto c\}, g\ x) = ([c \to e/b], e) \\
&\qquad\qquad T \quad\ = \mathcal{U}(Ss, s' \to d) = \mathcal{U}(a, e \to d) = [e \to d/a] \\
\mathcal{W}(\{f \mapsto a, g \mapsto b, x \mapsto c\}, g\ x) &= (T'S'R', T'e) = ([c \to e/b], e) \\
&\quad \text{where} \quad (R',u) = \mathcal{W}(\{f \mapsto a, g \mapsto b, x \mapsto c\}, g) = (\varnothing, b) \\
&\qquad\qquad (S',u') = \mathcal{W}(R'\{f \mapsto a, g \mapsto b, x \mapsto c\}, x) = (\varnothing, c) \\
&\qquad\qquad T' \quad\ = \mathcal{U}(S'u, u' \to e) = \mathcal{U}(b, c \to e) = [c \to e/b]
\end{aligned}
$$

List of Figures