Charles University, Prague

Faculty of Mathematics and Physics

# Master Thesis

Author: Jaroslav Tulach
Supervisor of Thesis: RNDr. Jan Hric
Major: Computer Science
Year: 1998

# Typing Theory in Terms of Graphs

Jaroslav Tulach

April 21, 1998

# Acknowledgements

First of all I would like to claim that all the type graph theory presented in this thesis is my work. All definitions, lemmas, theorems and proofs are completely written by me. I would like to ensure that I have never seen similiar attempt of using graphs for description of types. Of course the Hindley/Milner theory presented in the chapter one is not developed by me, but the type graph theory used to describe it is mine. The content of chapter two is not directly based on any other research and the requirements for types with subtyping are taken from common knowledges. Again the graph typing theory build to describe types with subtyping is created by me.

I declare that I have written whole thesis without anybody else help except following people, who I would like to thank. Without them the thesis would never exist or at least would be even worse.

**Jan Hric** guided my work, read everything I wrote. Learned my theory and corrected my definitions, lemmas and proofs. Found a lot of mistypings, oversights in definitions and bugs in proofs. Without his help the theory would have much more bugs.

**Helena Stolka, Jack Catchpoole** helped me to correct my English. Added a lot of *a* and *the* and rewrite many sentences from my English to the real one.

**Vlado Majerech** helped me with non-trivial LaTeX functions.

Thanks you all for your help. It has been pleasure to be supported by you. At last I would like to ensure everybody that I agree with the public distribution of this thesis.

Prague, 21th April 1998 Jaroslav Tulach

# Contents

# List of Figures

# List of Algorithms

# Introduction

People are used to thinking in cathegories. They like to sort things into groups. Their decisions are based on looks, tastes, sounds, etc. A set of things with common properties forms a type. This type collects things into a group with the same or similiar behaviour.

Programming languages also work with objects. Characters, numbers and strings are basic types one can encounter in nearly every computer language. The type not only determines the set of values, but also the set of operations that can be executed. Usually the set of types is not fixed but can be extended by a programmer. The way new types can be created and how types can be composed depends on the model of typing theory the language uses. That is why typing theories have been receiving a lot of attention.

The possibility to partially check correctness of a program and discover keying mistakes, oversights and evident nonsenses before the program is executed and crashes provides great motivation for prior and future research. Fundamentals have been based on the study of lambda-calculus. These are greately explained in [1]. Theories based on the original by Hindley/Milner have been used in many functional languages. The typical features of most functional languages include polymorphic types, treating functions as regular objects and often some kind of overloading. The functional languages are runtime safe. When a program is correctly typed it will run without any runtime errors.

The other motivation for the study of typing theories is based on computer languages as Algol, C, Pascal and object-oriented languages as Simula, Eiffel and Java. The object-oriented languages lack ways for manipulating methods but offer subclassing with the possibility to easily reuse and modify already written code. The OO languages safety is limited compared to functional languages. Without functional polymorphism the programs must use type castings. Such programs then cannot be fully checked for correctness and can fail on execution.

The actual question is wheter we could take advantage of subtyping that is very powerful for describing real applications like simulations and window management, and create a typing theory that will not lack casting problems. Also, we would like to take the good from functional languages and allow functions to be treated as other objects, and introduce polymorphic types to object oriented theory.

The work in this thesis is motivated by the Term Graph Rewriting System [3] designed to describe computation of rewriting systems. We take their model,

modify it to express not the computation process but the relation between types and provide modification operations. We describe types by nodes and edges in the graph. We give a precise description of what a type is by restriction of the set of type graphs to only those which represent meaningful types. We define how to compare two types and present algorithms working on such graphs.

The graph theory we try to develop in this thesis should provide new view on already known typing theories and should connect the world of graphs with the area of types. This could allow application of well-known graph algorithm to computation of types. We study how the graph theory can be used to express types with subtyping relationship also.

In the first chapter we present the original Hindley/Milner typing theory in terms of graphs. We define type graphs, provide ways of comparing types and introduce the main operation for modification of graphs, substitution. We define the set of equalities and provide an algorithm that can solve the set and find the best possible solution. We prove that the algorithm is finite and correct.

In the second chapter we show how the graphs can be extended to express subclassing. We give natural requirements for the creation of new constructors, describe restrictions to the set of constructors, present an algorithm for comparing types and discuss what the graph model is missing in order to provide a full working range for types with subclassing.

# Chapter 1

# Typing with graphs

The description of the typing theory with type variables is given in this chapter. First of all the examples common for functional languages are presented. Then the original Hindley/Milner theory is described in terms of graphs.

We show that the graphs provide practical way for representation of types. By describing the types by graphs we can use common graph algorithms for testing of cycles and etc. Also due to the choosen representation of variables our model does not have problems during substitutions. The variables need not be renamed, we need not check whether a variable is free for substitution or not.

The process of development the type graph theory goes as follows. We define what a type constructor is, describe how types are created from constructors and how they are represented by type graphs. Then substitution, our only operation on type graphs, is introduced. It allows us to define ways for comparing type graphs and after defining sets of equalities to specify what a solutions are and to choose the best solution of them.

Then we present an algorithm for solving sets of equalities. We prove that if there is a solution the algorithm finds the best one and that the algorithm finishes its computation for any input.

## 1.1 Types in Functional Languages

This section presents the implementation of the Hindley/Milner types in functional languges. Examples of expressions and types are presented and then the sence of constructors is described and their formal definition is given.

A usual program in common functional language consists of declarations of functions or variables. Each such object must have its type. The type can be specified explicitly or derived by the compiler. Each language comes with predefined basic types but also allows construction of new ones by definition of new type constructors.

The standard set of constructors usualy includes constructors for basic types

`Int`, `Real`, `String`, list constructor `[]` and function constructor `->`. Moreover a set of constructors `(,)`, `(,,)`, `(,,,)` and etc. is often provided to describe tuple, triple, quadruple and so on. Each constructor has assigned *arity*, the number of types that must be provided to it to form a new type. The constructors `Int`, `Real`, `String` have arity zero. That is why they do not need no other types, but immediately represent types. List constructor has arity one, so it needs one type to create new one. Examples of list types are list of integers `[Int]`, list of reals `[Real]` or list of lists of integers `[[Int]]`. The constructor `->` has arity two and the type it creates is used to represent a function.

**Example 1.1.1** In the following program (written in functional language Clean [2]) the predefined type *Int* is used together with the list constructor [] to create the list of integers type [*Int*]. Functional constructor $\rightarrow$ composes two types into one function type [*Int*] $\rightarrow$ *Int*.

```
// Function that takes list of integers and
// returns the last element in the list.
last :: [Int] -> Int         // definition of type of last
last [ x : y ] = last y
last [ x ] = x
```

The function `last` does not depend on integers. It need not know the element type it works on. It could without any modification work on *Real* or *String*. For such cases, the Clean language offers polymorphic *type variables*. The type of last could be `last :: [x] -> x`. The $x$ is a type variable (begins with lowercase) for which any concrete type can be substituted. So it is possible to use `last` on integers and also on strings (see 1.1.2 ).

**Example 1.1.2** Usage of polymorphic function `last`. In this example the type of `last` is `[x] -> x`.

```
last [1, 2, 3]        // produces integer 3
last [1.3, 2.77, 3.0] // produces real 3.0
last ["Hello", "Hi"]  // produces string "Hi"
```

**Example 1.1.3** Following piece of code defines new constructor which riches the set of default ones. The new constructor is named `Either` and has arity two. So when it is used (`Either a a`) in type definition, two types (in this case variables) must be provided.

```
// Definition of new type constructor
::Either a b = First a | Second b

// Function that returns the element
takeElement :: (Either a a) -> a
takeElement (First x) = x
takeElement (Second x) = x
```

In spite of that one can create own constructors, they are created during compilation. But their set does not change during the execution of the program. We state this immutability in following definition.

**Definition 1.1.1** The *set of constructors* can be any (possibly infinite) set. Let us denote it as $C$, and suppose that it is immutable through rest of this chapter.

(i) The function $arity : C \to \mathcal{N}$ assignes to each constructor a number of parameters needed to create a new type.

(ii) $C_k$ is a set of constructors with arity $k$ as defined by $C_k = \{c \in C : arity(c) = k\}$.

## 1.2 Graphs

In this section graphs are described. The graphs consist of nodes with assigned constructors symbols, oriented edges between nodes and list of important nodes, so called roots. We define special constructor that is used to mark nodes representing variables. Then we introduce the term direction that allow us to describe and work with the topology of the graph. Also few related terms based on the direction are specified.

The motivation for using graphs for describing types is taken from [3]. The Term Graph Rewriting is discussed there and graphs are used to describe functions, their arguments and variables. The model presented here is similiar but it describes types, their parameters and type variables. Also the operations on our graphs are different.

The *graph* consists of nodes and directed edges. To each node a constructor symbol is assigned. Its arity defines the number of outgoing edges from the node. The precise definition follows:

**Definition 1.2.1** *Graph* $g$ is quarduple $(V, symb, args, roots)$ where:

(i) $V$ is any finite set,

(ii) $symb : V \to C$ is a function that assigns a constructor to each node,

(iii) $args : V \to V^*$ defines outgoing edges such that for each $v \in V$ the following condition is satified: $arity(symb(v)) = length(args(v))$,

(iv) $roots \in V^*$ is a list of important nodes (roots) in the graph.

**Notation 1.2.1** When graph $g = (V, symb, args, roots)$ will be discussed, the symbols $V_g$, $symb_g$, $args_g$ and $roots_g$ can be used instead of $V$, $symb$, $args$ and $roots$ to distinguish the used graph.

**Notation 1.2.2** Whenever arguments of a node are quantified we use general quantifier instead of bounded one. So $(\forall i)\ args_g(u)_i$ is a shorter version of $(\forall i < length(args_g(u)))\ args_g(u)_i$.

To create a graph whose nodes would represent the Clean's type, we have to create separate nodes for all variables and assign a special constructor with zero arity to them. Let us denote it as $\perp$. Then we take the constructor in the


Figure 1.1: Representation of type [Int]

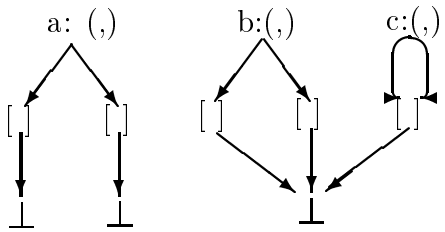


Figure 1.2: More types

type that uses only types whose nodes have already been created and create a new node for it. The outgoing edges are redirected to the used nodes.

**Definition 1.2.2** The *set of variables* for graph $g$ is denoted as $Var_g$ and defined as $Var_g = \{v \in V_g : symb_g(v) = \perp\}$.

The type `[Int]` can be represented by graph pictured at 1.1 . But one type can have more graph representations. For example `([u], [u])` can be represented as both roots $b$ and $c$ of graph at figure 1.2 . But the root $a$ represents type `([v], [w])`.

Usually each type in functional language has finite representation. So, by using the construction described above, we can construct acyclic graph for each type. Moreover, no cyclic graph represents any functional type.

**Example 1.2.1** The root $r$ of graph at figure 1.3 is an example of node not representing valid type. The type cannot be valid because it would have to become nonfinite `(a, (a, (a, ...)))`.

For this reason we restrict the set of graphs to acyclic. To achieve this the notion of *direction* is provided. It describes the sequence of indexes determining argument edges that should be choosen on the *path* from a node.

**Example 1.2.2** In the figure 1.4 the directions `(0, 0)` and `(1, 0)` lead from root $r$ to the node $a$ and the directions `(1, 1)` and `(2, 0)` go from root to the node $b$.

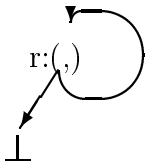


Figure 1.3: Cyclic type

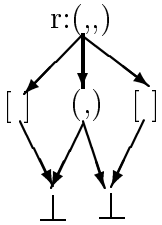


Figure 1.4: Directions

**Definition 1.2.3** Let $g$ be a graph. Let $v \in V_g$. Then
(i) *Direction* is any element of $Dir = \mathcal{N}^*$ (any sequence of natural numbers),
(ii) *Path along direction* $s \in Dir$ is any $p \in {V_g}^*$ where $length(p) = length(s) + 1$ and for each $i < length(s)$ $p_{i+1} = args_g(p_i)_{s_i}$,
(iii) when there is path $p$ along direction $s$ we say that the *direction* $s$ *leads* from $p_0$ to $p_{length(p)-1}$ and denote it as $p_0 \overset{s}{\rightsquigarrow}_g p_{length(p)-1}$,
(iv) $s$ is a *possible direction from* $v$ if there exists a node $u \in V_g$ such that $v \overset{s}{\rightsquigarrow}_g u$,
(v) symbol $Dir_g(v)$ denotes *all possible directions from node* $v$ *in graph* $g$ and is defined as $Dir_g(v) = \{s : (\exists u \in V_g)\ v \overset{s}{\rightsquigarrow}_g u\}$,
(vi) if $s \in Dir_g(v)$ then result of *application of the direction to node* $v$ is such node $u$ that $v \overset{s}{\rightsquigarrow}_g u$. The node $u$ can be then denoted by $s_g(v)$.

**Definition 1.2.4** Let $g$ be a graph. Node $v \in V_g$ is said to be *in a cycle* if there exists a direction $s$ such that $length(s) \geq 1$ and $v \overset{s}{\rightsquigarrow}_g v$.
(i) $g$ is *cyclic graph* if $(\exists v \in V_g)$ $v$ is in a cycle.
(ii) $g$ is *acyclic graph* otherwise.

**Example 1.2.3** Each node in graph represents a Clean's type. The root $r$ represents $([x], [y])$, nodes $u$ and $v$ represents $([z])$ and leaves $a$ and $b$ stand for a variable in the example 1.5 . But the type $([x], [y])$ is not specified only by node $r$. Because its edges lead into $u$, $v$ and indirectly into $a$ and $b$, these nodes are also needed. We say that they are *used* by $r$.

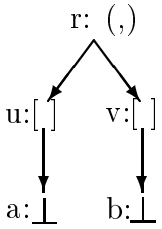


Figure 1.5: Each node represents type

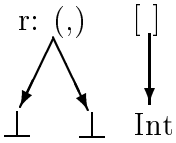


Figure 1.6: Nonreachable node

**Definition 1.2.5** Let $g$ be a graph. For each node $v \in V_g$ the *set of used nodes* can be defined as $Uses_g(v) = \{u : (\exists s \in Dir)\ length(s) \geq 1 \wedge v \overset{s}{\rightsquigarrow}_g u\}$.

The *path* and the *used nodes relation* are transitive. The following proposition specifies it more exactly.

**Proposition 1.2.3** Let $g$ be a graph, $u, v, w \in V_g$ nodes and $s_1$ and $s_2$ be directions such that $u \overset{s_1}{\rightsquigarrow}_g v$ and $v \overset{s_2}{\rightsquigarrow}_g w$

(i) there exists a direction $s$ that $u \overset{s}{\rightsquigarrow}_g w$,
(ii) $Uses_g(u) \supseteq Uses_g(v) \supseteq Uses_g(w)$.

The picture 1.6 shows graph where all nodes are not accessible from root $r$. This is still valid graph but in some cases (as in definition 1.6.1 ) we need some nodes to be reachable. That is why we present following useful definition.

**Definition 1.2.6** Let $g$ be a graph. The node $n \in V_g$ is said to be *reachable* if there is a root $r \in roots_g$ and direction $s \in Dir$ such that $r \overset{s}{\rightsquigarrow}_g n$.

## 1.3 Type graphs

We have defined graphs and presented terms like direction, used nodes and reachable node. In this section we restrict set of graphs to only such ones that represent valid types. Then the term of subgraph is defined.

**Definition 1.3.1** The graph $g$ is a *type graph* if it is acyclic. The set of all type graphs is denoted by $G$. Through the rest of the chapter we work only with graphs from this set.

As illustrated earlier the meaning of one node (and its subtree) and the whole graph with the node as a root are strongly related. To make the definition more specific we introduce the term *subgraph*, the part of graph under a node rooted in that node.

**Definition 1.3.2** Let $g$ be a graph and $v$ its node ($v \in V_g$). Then graph $g \mid v = (V, symb, args, roots)$ is a *subgraph* of $g$ at $v$. The components $V$, $symb$, $args$ and $roots$ are defined as follows:

(i) $V = \{u : (\exists s \in Dir)\ v \overset{s}{\rightsquigarrow}_g u\}$,
(ii) $symb = symb_g \mid V$ (the function $symb_g$ restricted only to range $V$),
(iii) $args = args_g \mid V$,
(iv) $roots = \langle v \rangle$ (vector with only one node $v$).

**Notation 1.3.1** Let $g$ be a graph. Let $u \in V_g$. The subgraph of $g$ at $u$ that is denoted as $g \mid u = (V_{g|u}, symb_{g|u}, args_{g|u}, roots_{g|u})$ can also be labeled as $(V_u, symb_u, args_u, roots_u)$.

Because the subgraph of $V$ at $v$ contains all nodes accessible (there exists a direction) from $v$, it is easy to prove that each path in a subgraph is also the path in the original graph.

**Proposition 1.3.2** Let g be a graph. Let $u \in V_g$. Then $(\forall v, w)\ v \overset{s}{\rightsquigarrow}_{g|u} w \Rightarrow v \overset{s}{\rightsquigarrow}_g w$.

From this proposition it immediatelly follows that when the original graph is acyclic the subgraph remains acyclic as well.

**Conclusion 1.3.3** Let $g$ be acyclic graph. Let $v \in V_g$. Then $g \mid v$ is acyclic.

## 1.4 Comparing types

In the section about *Type graphs* we could see that one type can have several graph representations. Therefore it is not sufficient to simply compare sets of nodes, their symbols, edges and roots to recognize whether two graphs represent the same type. The relationship between graphs is a little more complicated as described in this section.

To verify that two graphs represent the same type we first of all focus our attention on the most simple case. We try to compare two graphs with exactly one root.

**Definition 1.4.1** Let us have two graphs $g = (V_g, symb_g, args_g, \langle u \rangle)$ and $h = (V_h, symb_h, args_h, \langle v \rangle)$. Each has exactly one root. Then these *graphs are isomorphic* via function $f : Var_g \to Var_h$, denoted by $g \simeq_f h$ if all following conditions are satisfied:

(i) $Dir_g(u) = Dir_h(v)$,
(ii) $(\forall s \in Dir_g(u))\ symb_g(s_g(u)) = symb_h(s_g(v))$,

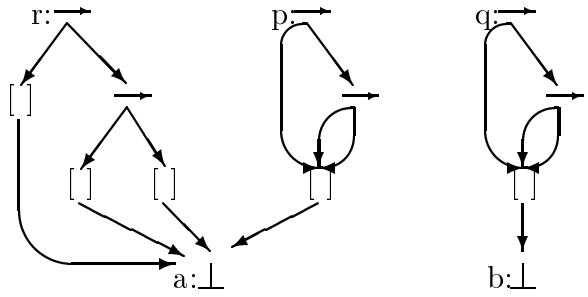


Figure 1.7: Two equivalent one-root graphs

(iii) $f$ is isomorphism
(iv) $(\forall w \in Var_g)(\forall s \in Dir_g(u))\ u \overset{s}{\rightsquigarrow}_g w \Leftrightarrow v \overset{s}{\rightsquigarrow}_h f(w)$.
These graphs are called *equivalent* if they are isomorphic via identity function. We use $g \equiv h$ to express the equivalency.

The definition declares that graphs $g$ and $h$ are isomorphic if their roots have the same set of possible directions. The same constructor is reached by taking the same direction from first and second root node. Variables from $Var_g$ and $Var_h$ are associated to pairs and when the same a direction leads to a variable from the root of first graph the same direction leads from root of graph $h$ to the associated variable. The difference between isomorphism and equivalence is that variables of both graphs in equivalence must be the same.

**Example 1.4.1** Graphs rooted in $r$ and $p$ on picture 1.7 have different sets of nodes but they represent the same type. Because sets of possible directions $Dir_g(r) = \{\lambda, (0), (0,0), (1,0), (1,0,0), (1,1,0)\} = Dir_g(p)$, node symbols $symb_g(s_g(r)) = symb_g(s_g(p))$ for each $s \in Dir_g(r)$ and all directions leading to variable node with symbol $\perp$ (that means $(0,0)$, $(1,0,0)$, $(1,1,0)$) reach the same node from both roots $p$ and $r$. The third graph with root $q$ is not equivalent because it used different variable but is isomorphic via function $f(a) = b$.

The equivalence on one-root graphs and the way the subgraph can be obtained from a node allow us to compose the definition for equvialence on nodes.

**Definition 1.4.2** Let $u, v$ be nodes in graph $g$. The nodes $u$ and $v$ are *equivalent*, that is denoted by $u \equiv_g v$ if $g \mid u \equiv g \mid v$.

We have a way for comparing nodes and we can extend the definition from one-root graphs to all type graphs. Two graphs will be considered equivalent if all their roots will be equivalent.

**Definition 1.4.3** Let $g$ and $h$ be graphs. They are said to be
(i) *equivalent* if $length(roots_g) = length(roots_h)$ and for each $i$ the $i$-th roots are equivalent. Formally $(\forall i < length(roots_g))\ g \mid (roots_g)_i \equiv h \mid (roots_h)_i$,

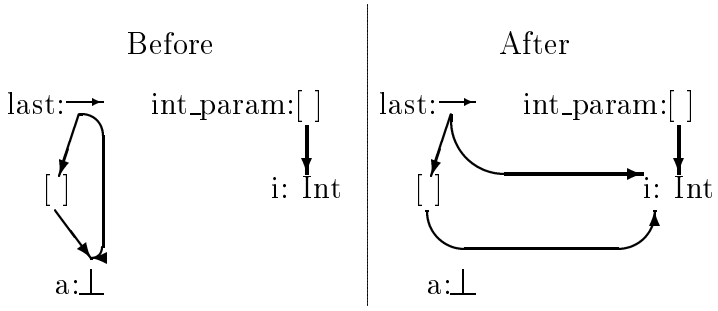


Figure 1.8: Substitution of $[Int]$ as the first param of function *last*

(ii) *isomorphic* if there exists a function $f : Var_g \to Var_h$ such that for each $i < length(roots_g)$ $g \mid (roots_g)_i \simeq_f h \mid (roots_h)_i$.

This definition is correct with respect to the definition of equivalence for one-root graphs because the definition 1.4.3 uses one-root graph definition to compare each root pair. Also, we can easily see that both relations defined in this section are equivalences.

**Proposition 1.4.1** The relations $\equiv$ and $\simeq$ are equivalences. For each graph $g$ the relation $\equiv_g$ is also equivalence.

## 1.5 Substitution on graphs

The function `last` presented in 1.1.1 has type `[a] -> a`. As is shown in example 1.1.2 , lists of integers, reals and other types can be passed as its parameter. Depending on the input parameter, the result is then integer or real. The process during which the type variable $a$ is associated with *Int* or *Real* is called substitution.

**Example 1.5.1** The picture 1.8 shows what happens when Int is substituted instead of variable $a$. All edges leading to node $a$ are redirected to the node $i$. Redirections are valid only on edges to variable nodes ($symb_g(n) = \perp$).

Therefore substitution could be described by function $f : Var_g \to V_g$ that could assign a new node to each variable. Such a partial function would complicate their composition and for this reason we will represent it as function with full domain $V_g$ but restrict that for each $n \notin Var_g$ it must be constant.

**Definition 1.5.1** Let $g$ be a graph. Let $f : V_g \to V_g$ be a function.
(i) $f$ is called *substitution on* $g$ if $\{v \in V_g : f(v) \neq v\} \subseteq Var_g$,

(ii) result of applying substitution $f$ to graph $g$ is graph denoted by $f(g) = (V_g, symb_g, args, roots)$ where

$$\begin{array}{lllll} (\forall v \in V_g)(\forall i) & args(v)_i & = & f(args_g(v)_i) \\ & length(roots) & = & length(roots_g) \\ (\forall i) & roots_i & = & f((roots_g)_i) \end{array}$$

(iii) substitution $f$ is called *valid* if the graph $f(g)$ is acyclic.

Substitutions can be composed. One can be applied after another. Because they are represented by functions, these functions can be composed to one too. Is the resulting function also a substitution? Is the result of applying all substitutions and the composed one the same? The following lemma answers these questions.

**Lemma 1.5.1** Let $g$ be a graph. Let $f_1$, $f_2$ be substitutions on $g$. Then
(i) $f_1$ is a also substitution on graph $f_2(g)$,
(ii) $(f_1 \circ f_2)$ is a substitution on $g$,
(iii) $(f_1 \circ f_2)(g) = f_1(f_2(g))$.

*Proof.* First of all let us notice that even if the second substitution uses variables that already have been substituted by the first substitution, there are no problems with it. The first substitution redirects edges leading in the original graph to substituted variables to different nodes in the mid-resulting graph and that is why the second substitution has no affect on these variables. This seems to be one of the great advantages of our theory. Keep it in mind and let us provide exact proof for each step.

(i) The only requirement for a function on graph nodes to be substitution is that the function is constant on nonvariable nodes. Because $V_{f_2(g)} = V_g$ and $Var_{f_2(g)} = Var_g$, it is easy to obtain that $\{v : f_1(v) \neq v\} \subseteq Var_g = Var_{f_2(g)}$. The $f_1$ is substitution on graph $f_2(g)$,

(ii) Let $f = (f_1 \circ f_2)$. The set $\{v \in V_g : f(v) = v\} \supseteq \{v \in V_g : f_1(v) = v \wedge f_2(v) = v\}$. From this we get the result that $\{v \in V_g : f(v) \neq v\} \subseteq \{v \in V_g : f_1(v) \neq v\} \bigcup \{v \in V_g : f_2(v) \neq v\} \subseteq Var_g$. For this reason $f_1 \circ f_2$ is substitution on $g$.

(iii) Let $h_1 = (f_1 \circ f_2)(g)$ and $h_2 = f_1(f_2(g))$. Then evidently $V_{h_1} = V_g = V_{h_2}$ and $symb_{h_1} = symb_g = symb_{h_2}$. The i-th root changes to $f_1(f_2(roots_{g\,i})) = (f_1 \circ f_2)(roots_{g\,i}))$. Therefore the only thing that is not so clear is the equivalence of $args_{h_1}$ and $args_{h_2}$. Let $v \in V_g$. Then $args_{h_1}(v) = (f_1 \circ f_2)(args_g(v)) = f_1(f_2(args_g(v)))$. In the same manner we get $args_{h_2}(v) = f_1(args_{f_2(g)}(v)) = f_1(f_2(args_g(v)))$. So, it is right that $h_1 = h_2$. □

The substitutions can be used for comparing the generality of graphs. We can say that one graph is more general than the other if the former represents a type which can create the latter type by a substitution.

**Definition 1.5.2** Let $g$ and $h$ be graphs.

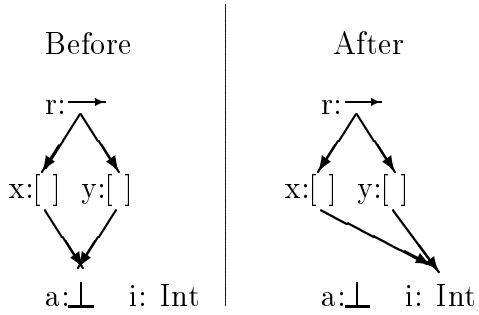


Figure 1.9: Substitution of f(a) = i

(i) We say that $g$ is *more general* than $h$ via substitution $f$ and denote it by $g \rhd_f h$ if $f$ is valid substitution on $g$ and $h \equiv f(g)$,
(ii) graph $g$ is said to be more general than $h$ (notated as $g \rhd h$) if there exists a substitution $f$ such that $g \rhd_f h$.

There is a relationship between substitution and equivalence. Whenever two nodes in a graph are equivalent, they will remain equivalent in any graph that can be obtained by a substitution. The picture 1.9 shows that equivalent nodes $x$, $y$ remain equivalent after applying substitution $f(a) = i$. The following lemma claims that this behaviour is common for all substitutions in this chapter.

**Lemma 1.5.2** Let $v_1$ and $v_2$ be nodes in graph $g$. Let $f$ be a valid substitution on $g$. Let graph $h = f(g)$. Then $v_1 \equiv_g v_2 \Rightarrow f(v_1) \equiv_h f(v_2)$.

*Proof.* At first we should realize that if there is a direction $s = \langle s_0, \ldots, s_k \rangle$ leading to node $w \in V_h$ by means of $v_1 \overset{s}{\rightsquigarrow}_h w$, then there are two possibilites:

(i) if there exists $l \leq k$ and $u \in Var_g$ such that direction $\overline{s} = \langle s_0, \ldots, s_l \rangle$ leads in graph $g$ from $v_1$ to node $u$, then by following the direction $\overline{s}$ in graph $h$ we get into $f(u)$. So $f(v_1) \overset{\overline{s}}{\rightsquigarrow}_h f(u)$. Because $v_1 \equiv_g v_2$ the direction $\overline{s}$ from $v_2$ in graph $g$ leads into $u$ too and the result is that $f(v_2) \overset{\overline{s}}{\rightsquigarrow}_h f(u)$. It is obvious that by concatenting the rest of direction $s$ to $\overline{s}$, we get $f(v_2) \overset{s}{\rightsquigarrow}_h w$. The direction from both $f(v_1)$ and $f(v_2)$ leads to the same node.

(ii) if there is no such $l$ then $v_1 = f(v_1)$, $v_2 = f(v_2)$ and also in graph $g$ $v_1 \overset{s}{\rightsquigarrow}_g w$. Follow in the direction $s$ from $v_2$. There is a node $u$ such that $v_2 \overset{s}{\rightsquigarrow}_g u$. It is not necessary that $u = w$, but because $v_1 \equiv_g v_2$ we know that $symb_g(w) = symb_g(u)$, and because $u, w \notin Var_g$ we can be sure that $symb_h(w) = symb_h(u)$.
From both posibilities mentioned above, we get $Dir_h(f(v_1)) = Dir_h(f(v_2))$, that for each direction the symbols on the path from $v_1$ and $v_2$ are the same and that when a direction leads to a variable from $v_1$, it leads to the same variable from $v_2$. The conclusion is that $f(v_1) \equiv_h f(v_2)$. □

In conclusion, we prove lemma that discovers behaviour of $\rhd$ relation.

**Lemma 1.5.3** The relation $\rhd$ is reflexive and transitive.

*Proof.* Let $g$ be any acyclic graph. We can define function $id(x) = x$ for each $x \in V_g$. Then $g = id(g)$ and $id$ is a valid substitution on $g$. That is why $g \rhd g$.

Let $g_1 \rhd g_2$ and $g_2 \rhd g_3$. Then, there are valid substitutions $f_1$ and $f_2$ such that $g_2 \equiv f_1(g_1)$ and $g_3 \equiv f_2(g_2)$. We know that $f_1 \circ f_2$ is also a valid substitution on $g_1$ and that $(f_1 \circ f_2)(g_1) = f_2(f_1(g_1)) \equiv f_2(g_2) \equiv g_3$. That is why $g_1 \rhd g_3$ via $f_1 \circ f_2$ and the transitivity condition of $\rhd$ is satisfied.

The relation between $\rhd$ and $\simeq$ is expressed by following lemma.

**Lemma 1.5.4** Let $g$ and $h$ be graphs. Let $g \rhd h$ and $h \rhd g$. Then there exists a function $f : Var_g \to Var_h$ such that $g \simeq_f h$.

*Proof.* There are substitutions $l, k$ such that $g \rhd_l h$ and $h \rhd_k g$. The first step is to prove that functions $l$ and $k$ must map each reachable variable node to another variable node.

Suppose that there is a direction $s$ leading from $i$-th root of graph $g$ (let us denote it as $r_{g,i}$) to a variable $s_g(r_{g,i}) \in Var_g$ and that the same direction does not lead to any variable from $i$-th root of graph $h$. $s_h(r_{h,i}) \notin Var_h$. The graphs have the same set of variables $Var_g = Var_h$ because they are related by $\rhd$.

The node $n = s_h(r_{h,i})$ is not variable. That is why the substitution $k$ cannot alter it, so $k(n) = n = s_{k(h)}(r_{k(h),i})$. As a result $symb_{k(h)}(s_{k(h)}(r_{k(h),i})) \neq \perp$. Because $h \rhd_k g$ the graphs $k(h) \equiv g$ and $symb_g(s_g(r_{g,i})) \neq \perp$ but this cannot be true. We supposed something else. For this reason for each $v \in Var_g$ and $v$ is reachable $l(v) \in Var_h$. The reverse direction for the substitution $k$ is obviously true too.

The second part of the proof is to test whether the substitutions $l$ and $k$ are monomorphic on reachable variables.

Suppose that there are directions $s$ and $t$ leading to different variables from any roots in graph $g$. $s_g(r_{g,i}), s_g(r_{g,j}) \in Var_g = Var_h$ and $s_g(r_{g,i}) \neq s_g(r_{g,j})$. In the graph $h$ these directions must lead to variables, suppose that they lead to the same variable $s_h(r_{h,i}) = s_h(r_{h,j}) \in Var_h$. Then after applying the substitution $k(s_h(r_{h,i})) = k(s_h(r_{h,j}))$ and for this reason $s_g(r_{g,i}) = s_g(r_{g,j})$. This contrasts with our assumption. We proved that for each pair of different rechable variables $u, v \in Var_g$ the $l(u) \neq l(v)$. Similiarly for the second substitution.

Finally we define function $f : Var_g \to Var_h$ as

$$f(u) = \begin{cases} l(u) & \text{if } u \text{ is reachable in graph } g \\ v & \text{if } k(v) = u \text{ and } v \text{ is reachable in graph } h \\ u & \text{otherwise} \end{cases}$$

Even the first two conditions can occur together, we can be sure that the definition of function $f$ is correct because graphs $g \equiv k(l(g))$ and for each $v \in V_g$ the $k(l(v)) = v$. The proof is done because $g \simeq_f h$. $\square$

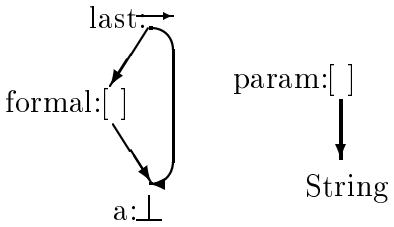


Figure 1.10: last ["Hello", "World"]

## 1.6 Node Equalities

A set of equalities on a graph $g$ is a set of node pairs. Each equality desire the node pair to be equivalent in some graph. We say that such graph $h$ is solution if it is less general then $g$ and all nodes in pairs are equivalent.

**Example 1.6.1** Equalities are created from any functional expresion. For example, the well know example with function `last ["Hello", "World"]` creates a graph on picture 1.10 . Since the type $[String]$ is applied as the first argument of function $last$, the set of equalities contains one pair $(formal, param)$,

**Definition 1.6.1** Let $g$ be a graph. The *set of equalities* is any pair $(g, M)$, where $M \subseteq \{u \sim v : u, v \; are \; reachable\}$.

The reason why nodes must be reachable from roots is that we want equivalent graphs ($g \equiv h$) to behave similarly. However, since the equivalence of graphs depends only on reachable nodes, we have to restrict the equalities to use only these nodes.

When a substitution is applied on a graph, a new graph is produced. Also, when applied on a set of equalities, the equalities must change. The natural way is to convert node pairs by use of the substitution function.

**Definition 1.6.2** Let $(g, M)$ be a set of equalities. Let $f$ be a valid substitution on $g$. *Applying substitution* $f$ on $M$ is denoted as $f(M)$ and declared as $f(M) = \{f(u) \sim f(v) : u \sim v \in M\}$.

The new set of equalities $(f(g), f(M))$ is valid because all node pairs from $f(M)$ remain reachable in graph $f(g)$.

**Definition 1.6.3** Let pair $(g, M)$ be a set of equalities. Then

(i) graph $g$ is the *solution* of $M$ if $(\forall \; (u \sim v) \in M) \; u \equiv_g v$,

(ii) The *set of solutions* of $(g, M)$ is $Sol_g(M) = \{h : (\exists f) \; f \; is \; a \; valid \; substitution \; on \; g \; \wedge \; h \simeq f(g) \wedge \; f(g) \; is \; the \; solution \; of \; f(M)\}$,

(iii) graph $h$ is *the most general solution*, if for each solution $r \in Sol_g(M)$ it is true that $h \rhd r$.

The point (ii) of the definition guarantees that if a graph is a solution, then all isomorphic graphs are also solutions. Also, there can be more most general solutions but all of them are isomorphic.

**Proposition 1.6.1** Let $h_1, h_2 \in Sol_g(M)$ be most general solutions. Then $h_1 \simeq h_2$.

*Proof.* $h_1 \rhd h_2$ and $h_2 \rhd h_1$. From lemma 1.5.4 $h_1 \simeq h_2$.

The transitivity of $\rhd$ allow us to prove following lemma. As a result, we obtain that if there is a solution, all less general graphs are also solutions.

**Lemma 1.6.2** Let $h_1 \in Sol_g(M)$ and $h_1 \rhd h_2$. Then $h_2 \in Sol_g(M)$.

*Proof.* The $h_1 \rhd h_2$ implies that there is a substitution $f_1$ such that $h_2 = f_1(h_1)$. Also, because the $h_1 \in Sol_g(M)$, we know that there is substitution $f_2$ that $h_1 \equiv f_2(g)$. From that and lemma 1.5.1 we get that $h_2 \equiv f_2(f_1(g)) = (f_2 \circ f_1)(g)$. That is why $h_2 \in Sol_g(M)$. □

## 1.7 Algorithm for Solving Equalities

An algorithm is presented in this section that takes a graph and set of equalities, and finds out whether there is a solution, and if so, it finds the most general one. During processing the algorithm either fails, then there is no solution for the given graph and set of equalities, or it succeed and produces a graph. The graph is less general then the original given on input and satisfies all conditions from the set of equalities. The program is written in functional languages-like pseudo code extended with operations from Theory of Sets.

```
EqSolve :: (Graph, Set of Equalities) → Graph;
EqSolve (g, ∅) = g;                                              // case 0
EqSolve (g, {u ∼ v} ∪ M)
| u == v = EqSolve (g, M);                                       // case 1
| symb_g(u) == ⊥ ∧ u ∈ Uses_g(v) = FAIL;                         // case 2
| symb_g(u) == ⊥ = EqSolve (f(g), f(M))                          // case 3
where {
  f :: V_g → V_g;
  f u = v;
  f x = x;
};
| symb_g(v) == ⊥ = EqSolve (g, {v ∼ u} ∪ M);                     // case 4
| symb_g(u) <> symb_g(v) = FAIL;                                 // case 5
| otherwise = EqSolve (g, M ∪ ⋃_i{args_g(u)_i ∼ args_g(v)_i});   // case 6
```

Algorithm 1.1: Algorithm for solving of equations

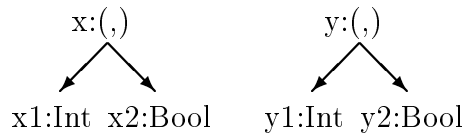


Figure 1.11: Equivalence of nodes by equivalence of subnodes

## 1.8 Correctness of the Algorithm

This section is dedicated to the proof that when $EqSolve(g, M)$ succesfully finishes, it really finds the most general solution of $(g, M)$, $M$ on $g$ and that when it fails there is no solution.

First of all we take a deeper look at the behaviour of relation $\equiv$ as defined in 1.4.2 . Two variable nodes are equivalent only if they are identical, since between equivalent nodes all variables must be shared. A proposition follows.

**Proposition 1.8.1** Let $g$ be a graph and $u, v \in Var_g$. Then $u \equiv_g v \Leftrightarrow u = v$.

There is another rule for nonvariable nodes. In the example 1.11 , nodes $x$, $y$ are equivalent because they have the same constructor symbol assigned and their subnodes $x_1$, $y_1$ and $x_2$, $y_2$ are equivalent. This behaviour is not bound only to this example, but is generic as is shown in the following lemma.

**Lemma 1.8.2** Let $g$ be a graph with nodes $u, v \notin Var_g$. Then $u \equiv_g v \Leftrightarrow symb_g(u) = symb_g(v) \wedge (\forall i)\ args_g(u)_i \equiv_g args_g(v)_i$.

*Proof.* "$\Rightarrow$" When we consider zero length direction and equivalence condition (ii) in the definition 1.4.1 , we see that $symb_g(u) = symb_g(v)$. We should then realize that for each $i$ and each direction $s$ from $args_g(u)_i$ or $args_g(v)_i$ there is appropriate direction $t = (i, s_0, \ldots s_{length(s)-1})$ such that $t_g(u) = s_g(args_g(u)_i)$ and $t_g(v) = s_g(args_g(v)_i)$. Because all equivalence conditions for direction $t$ are satisfied they are also satisfied, for direction $s$. Therefore, $args_g(u)_i \equiv_g args_g(v)_i$,

"$\Leftarrow$" The reverse direction is nearly same. We know that any direction $t = (t_0, \ldots t_k)$ from $u$ is also the direction from $v$ because $symb_g(u) = symb_g(v)$ (this implies that $|args_g(u)| = |args_g(v)|$). We also known that the first element of $t$ leads (from $u$ and $v$) to nodes that are equvivalent $args_g(u)_{t_0} \equiv_g args_g(v)_{t_0}$. As a result the rest of direction $(t_1, \ldots, t_k)$ satisfies the conditions for equvivalence and from this $u \equiv_g v$. □

The main goal of this chapter, the proof of correcteness of the algorithm $EqSolve$, is formulated in following theorem.

**Theorem 1.8.3** Let (g, M) be any set of equalities. Then

(i) when $Sol_g(M) \neq \emptyset$ $EqSolve$ finds a most general solution.

(ii) when $Sol_g(M) = \emptyset$ algorithm $EqSolve$ stops showing that there is no solution.

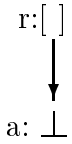


Figure 1.12: Used node

*Proof.* The *EqSolve* takes pair $(g, M)$ and either produces a result, fails or transforms the pair to another one $(h, N)$ and calls itself again with the new parameter.
In the proof, we study each case that can occur in the algorithm, and if it produces a result, we claim that it is a most general one. If it fails, we prove that $Sol_g(M) = \emptyset$. When it proceeds with recursion, we prove that $Sol_g(M) = Sol_h(N)$ for each step, so the set of solutions does not change during execution of algorithm 1.1 .
The proof is split into a few sections, each studying one case occuring in the algorithm.

## Case 0

`EqSolve (g, ∅) = g;`

This case is choosen when there are no equalities to solve. The $Sol_g(\emptyset) = \{h : g \rhd h\}$. The graph $g$ is the most general solution, and the algorithm produces the right result.

## Case 1

`EqSolve (g, {`$u \sim v$`} ∪ M) | u == v = EqSolve (g, M);`

For any $h \in Sol_g(M)$ node $u \equiv_h u$. Then $h \in Sol_g(\{u \sim u\} \cup M)$. We can omit equation $(u, u)$ because it does not affect the result.

## Case 2

`EqSolve (g, {`$u \sim v$`} ∪ M) |` $symb_g(u)$ `==` $\perp \wedge u \in Uses_g(v)$ `= FAIL;`

An example of this situation is drawn in figure 1.12 . It occures when a variable (in the example $a$) should be made equivalent to a node that uses the variable (node $r$). As shown by the following lemma, this is not possible until we allow cyclic graphs, and as a result the algorithm has the full right to fail.

**Lemma 1.8.4** Let $(g, \{u \sim v\} \cup M)$ be a set of equalities and let $u \in Var_g$ and $u \in Uses_g(v)$. Then $Sol_g(\{u \sim v\} \cup M) = \emptyset$.

*Proof.* Let $h \in Sol_g(\{u \sim v\} \cup M)$. Then, there is a substitution $f$ such that $f(g) \simeq h$. Because the node $v$ is not variable, $f(v) = v$. In the resulting graph after substitution, $f(u) \equiv_{f(g)} v$. From this, we see that $Dir_h(f(u)) = Dir_h(v)$.

Now let us take the longest direction that can be followed from $v$ and call it $t$. Because $f(u) \in Uses_{f(g)}(v)$, there exists a direction $s$ which $length(s) \geq 1$ such that $v \overset{s}{\rightsquigarrow}_g f(u)$. It is important to realize that when we take direction $s$ from $v$, we get to $f(u)$, but here we can continue along direction $t$. For this reason the concatenation $st \in Dir_{f(g)}(v)$. However, this conflicts with our assumption that $t$ is the longest possible direction. $f(g)$ is not a valid solution and because the assumed graph $h$ is equivalent with it, the graph $h$ is not valid solution either. The result is that $Sol_g(\{u \sim v\} \cup M) = \emptyset$.□

## Case 3

`EqSolve (g,` $\{u \sim v\} \cup$ `M) |` $symb_g(u)$ `==` $\perp$ `= EqSolve (f(g), f(M))`
`where f(x) = if (x == u) v x`

When the algorithm is about to solve equality of a variable and any other node $v$, it simply creates new substitution that assignes to the variable the node $v$. Other nodes are not substituted. The following lemma proves that this behaviour is sufficient because the new set of equalities has the same set of solutions as the original one.

**Lemma 1.8.5** Let $(g, \{u \sim v\} \cup M)$ be set a of equalities and let $u \in Var_g$ and $u \notin Uses_g(v)$. Then $Sol_g(\{u \sim v\} \cup M) = Sol_{f(g)}(f(M)))$ where

$$f(x) = \begin{cases} v & if\ x = u \\ x & otherwise \end{cases}$$

*Proof.*

"$\Leftarrow$" Let $h$ be a substitution such that graph $h(f(g)) \in Sol_{f(g)}(f(M))$. The $f(u) = f(v)$ holds because of the definition of substitution $f$. This equality implies that $f(u) \equiv_{f(g)} f(v)$ and from lemma about substitutions ( 1.5.2 )$h(f(u)) \equiv_{(h \circ f)(g)} h(f(v))$. That is why graph $(h \circ f)(g) \in Sol_g(\{u \sim v\} \cup M)$.

"$\Rightarrow$" Similary, $h$ will be a substitution and $h(g) \in Sol_g(\{u \sim v\} \cup M)$. For this reason $h(u) \equiv_{h(g)} h(v)$. To finish the proof it would be sufficient to claim that $h(g) \simeq (h \circ f)(g)$. It can be done by proving that for each $x, y \in V_g$ the $x \equiv_{h(g)} y \Leftrightarrow x \equiv_{(h \circ f)(g)} y$. From lemma 1.8.1 we see that if $symb_{h(g)}(x) = \perp$ then $x = y$, and the solution is correct. If the node $x$ is not variable, then from 1.8.2 we need if for each $i$ is satisfied $args_{h(g)}(x)_i \equiv_{h(g)} args_{h(g)}(y)_i$ to satisfy the same also in graph $(h \circ f)(g)$. Because the difference between $h$ and $h \circ f$ is only in node $u$

$$(h \circ f)(x) = \begin{cases} h(x) & if\ x \neq u \\ h(v) & if\ x = u \end{cases}$$

Either the edges in graphs $h(g)$ and $(h \circ f)(g)$ lead into the same node, and then the similiarity is clear or the edge in $h(g)$ leads to $u$ and in the graph $(h \circ f)(g)$ leads to $v$. Because subtrees $h(g) \mid u$ and $(h \circ f)(g) \mid v$ are equivalent there is no difference between their nodes. Therefore the whole graphs $h(g) \equiv (h \circ f)(g)$. □

## Case 4

$$\texttt{EqSolve (g, } \{u \sim v\} \cup \texttt{ M)}$$
$$\mid symb_g(v) \texttt{ == } \bot \texttt{ = EqSolve (g, } \{v \sim u\} \cup \texttt{ M);}$$

When a variable occurs on the second position in a equation, we rotate it and pass it again to the algorithm. Because the $\equiv$ is equivalence, the resulting set is the same.

Due to the semantics this case is activated only if $symb_g(u) \neq \bot$, otherwise the *case 2* or *case 3* are choosen.

## Case 5

$$\texttt{EqSolve (g, } \{u \sim v\} \cup \texttt{ M)} \mid symb_g(u) \texttt{ <> } symb_g(v) \texttt{ = FAIL;}$$

If an equation of two nonvariable nodes occures and they represent different constructor symbols, the set of equations does not have a solution. The proof is given in following lemma.

**Lemma 1.8.6** Let $(g, \{u \sim v\} \cup M)$ be set of equalities. Let $u, v \notin Var_g$ and $symb_g(u) \neq symb_g(v)$. Then $Sol_g(\{u \sim v\} \cup M) = \emptyset$.

*Proof.* Let $f$ again denote substutition and $h$ graph $h = f(g)$ such that $h \in Sol_g(\{u \sim v\} \cup M)$. Then $f(u) \equiv_h f(v)$ but because $u, v \notin Var_g$ then $u = f(u)$ and $v = f(v)$ but $symb_h(u) = symb_g(u) \neq symb_g(v) = symb_h(v)$. That is why $h$ is not a solution for the set of equalities. □

## Case 6

$$\texttt{EqSolve (g, } \{u \sim v\} \cup \texttt{ M)}$$
$$\mid \texttt{otherwise = EqSolve (g, M} \cup \bigcup_i \{args_g(u)_i \sim args_g(v)_i\}\texttt{);}$$

The last case occurs when an equation of two nodes with the same symbol is found. As is shown in the next lemma, the equation can be successfully substituted by equations on subnodes without any change to the set of solutions.

**Lemma 1.8.7** Let $(g, \{u \sim v\} \cup M)$ be set of equalities. Let $u, v \notin Var_g$ and $symb_g(u) = symb_g(v)$. Then

$$Sol_g(\{u \sim v\} \cup M) = Sol_g(M \cup \bigcup_i \{args_g(u)_i \sim args_g(v)_i\})$$

*Proof.* Again let $f$ denote a substutition and $h$ graph $h = f(g)$ such that $h \in Sol_g(\{u \sim v\} \cup M)$. Because neither $u$ or $v$ represents a variable $f(u) = u$ and $f(v) = v$. From lemma 1.11 we know that $u \equiv_h v \Leftrightarrow (\forall i)\ args_h(u)_i \equiv_h args_h(v)_i$. That is why we need to satisfy only equivalences of all arguments:

$$h \in Sol_g(M \cup \bigcup_i \{args_g(u)_i \sim args_g(v)_i\})$$

The proof of theorem 1.8.3 is finished. □

## 1.9 Finiteness of the Algorithm

In this section the algorithm 1.1 is proven to be finite for any given input. It is shown that after finite number of computational steps the algorithm either fails or successfully stops for any graph $g$ and set of equalities $M$.

The function $EqSolve$ has no loops and only recursion can cause infinite computation. To prove that this cannot happen, we present the following lemma with detailed proof based on computation of potential functions.

**Lemma 1.9.1** Let $M$ be a set of equalities on graph $g$. The computation of $EqSolve(g, M)$ will terminate after finite number of steps.

*Proof.* Let us define a few potential functions that assign a positive number to each graph $g$ and set $M$. The first function describes the state of set of equalites on graph $g$. It is defined as sum of the number of nodes in subgraphs of each elements in set of equalities. Exactly:

$$\Phi_1(g, M) = \sum_{u \sim v \in M} |V_{g|u}| + |V_{g|v}|$$

For each $g$ and $M$ $\Phi_1$ is bounded to $0 \leq \Phi_1(g, M) \leq 2|V_g||M| \leq 2|V_g|^3$ because the set $M \subseteq V_g \times V_g$. The second function describes the state of the graph and does not depend on set of equalities. It counts number of variable nodes reachable (there is a direction leading into the node) from any root of graph $g$. More exactly:

$$\Phi_2(g) = \#\text{ of reachable variables}$$

The function is also never negative. We show that for each of its cases 0–6 the algorithm 1.1 decreases at least one of these functions. As a result we see that the algorithm must finish.

• *case 0* – the algorithm does not continue in recursion and succesfully finishes,
• *case 1* – the recursion goes on but because $\Phi_1(g, \{u \sim v\} \cup M) > \Phi_1(g, M)$ potential $\Phi_1$ is decreased and $\Phi_2$ remains the same,
• *case 2* – the process fails and finishes,
• *case 3* – similiarly as in *case 1* the potential $\Phi_1$ is decremented and $\Phi_2$ does not change,
• *case 4* – there is no change in potential $\Phi_1$ or $\Phi_2$ but the $EqSolve$ is called with different arguments that guarantee that the next chosen case is either *case*

*3* or *case 2* and therefore either the function $\Phi_1$ is decremented or the algorithm finishes,
• *case 5* – the program stops unsuccessfully,
• *case 6* – potential $\Phi_1$ can grow but only up to $|V_g|^3$. The second potential $\Phi_2$ decreases by one, because for any substitution $f$, after applying substitution it the variable node $u$ is not accessible anymore.

To prove that the algoritm is finite, we now define a new potential function composed from previous two:

$$\Phi(g, M) = 2|V_g|^3 \ \Phi_2(g, M) + \Phi_1(g, M)$$

It is constructed in a way that guarantees that its value decreases in each case of algorithm $EqSolve$. As a result we can say that the function $EqSolve$ finishes for every argument. □

## 1.10 Conclusion

In this chapter, we have presented typing theory in terms of graphs. The theory is not new, but is given in new, unusual and precise terms. We have shown that the theory is strong enough to represent the polymorpic types introduced by Hindley/Milner.

We developed basics for comparing graphs to find out whether two graphs represent the same type and also whether a graph is more general than another. We defined equations on graph nodes, their solution and also the meaning of the most general solution.

We have shown the algorithm $EqSolve$ for solving sets of equations. We claimed that the algorithm is in all cases correct and finite and that if there is a solution, then it finds one of the most general ones.

# Chapter 2

# Subtyping

Types, subtypes and their hierarchy are discussed in this chapter. First of all the proposition of extension of the function language Clean is provided so Clean's record types are turned into objects types connected between each other by a super type relation.

After that we extend the definition of the graph from previous chapter to reflect the relation between types. We focus only on types derived from one common super type. We study the proposed features of the implementation and give the requirements on the constructors and set of types. We discuss the relation between different types and, as in the non-subtyping case where we used equalities, we define set of inequalities, we define what it means for the set to have a solution and provide an algorithm that can check whether a given graph satisfies conditions stated by the set of inequalities.

We also study the reasons why the graph model cannot be used to describe types with subclassing. The discussion shows the differences between Hindley/Milner and extended types.

## 2.1 Code reuse vs. subtyping

It is usual in object oriented languages like Smalltalk, C++ and Java that one can extend already defined type by adding some behaviour and create a new subtype that can be used anywhere the original can. But we should realize that there is a difference between extending for code reuse and extending for creation of subtype. Neither C++ or Java distinguish it. It results into few quaint situations.

**Example 2.1.1** For example the window toolkit classes `java.awt.Frame` and `java.awt.Button` extend base class for all visuals `java.awt.Component`. Any `Component` and its subclasses (e. g. `Button`) can be added to a frame. But can `Frame` be added to another `Frame`? No, it cannot. It is an example of subclassing that reuses the code but should not create a subtype. It is not

possible to express such behaviour in the Java language without writing a lot of useless code.

The focus of this chapter is not concerned on reuse of code neither are concerned implementation details. We focus on subtype relation theory that is not based on detailed knowledge of implementation. We are interested in studying the conditions which allow subtype to be used in place of supertype.

## 2.2 Records and Objects

There is a special type in the Clean language [2] that collects more named types into one (see program 2.1 ). The extension expressing relation between records should allow to specify which record the newly defined extends. It should also enable us to extend records parametrized with type variables. The syntax is shown at 2.2 .

```
// record with two reals
::Point = { x :: Real, y :: Real }
// record with type variables
::ElementAndList a = { element :: a, list :: [a] }
// record in record
::IntRealRecord = {
  i :: ElementAndList Int,
  r :: ElementAndList Real
}
```

Algorithm 2.1: Records in Clean

```
// type Circle that extends type Point and
// adds new field radius
::Circle = { Point & radius :: Real }

// simple object type
::SimpleType a b = { Object & first :: a, second :: b }

// composite derived type
::Type a b c = { SimpleType [a] (b -> c) & value :: [a] -> b}
```

Algorithm 2.2: Object extension to Clean

Unlike to the previous chapter the constructor is not specified only by its arity but also by its super type. Figure 2.1 describes how the type is connected to the super node. Dotted lines are used to assign the super type of nodes.

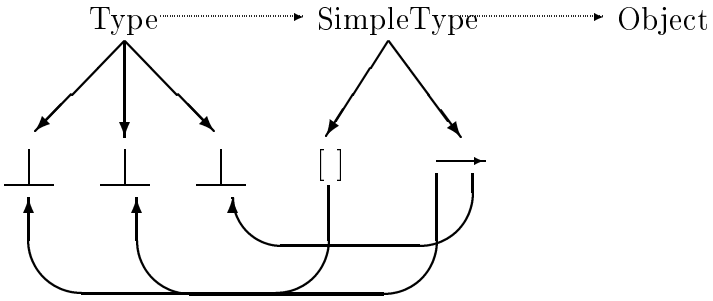


Figure 2.1: Type derived from SimpleType

We suppose that all types are derived from a basic type. Let us call it *Object*. It has no super type. But this is the only exception. All other types are required to have exactly one.

In usual object oriented languages the type has an immutable super type. We keep this model and extend it to types with type variables. Each constructor (except the *Object*) has a super type assigned. This type uses only variables provided by the constructor (as can be seen on figure 2.1 ). Dotted horizontal lines are used to express the super type relationship, full lines have the same meaning as in the previous chapter, describing type arguments.

Type with type variables can be used to form less general type by substitution of concrete types. The super type then changes appropriately. For example `Type Int Real Char` extends `SimpleType [Int] (Real -> Char)`.

## 2.3 Extended Graphs

In this section the definition of the graph from the previous chapter is modified to express the superclass relationship between types. A graph for super type is assigned to each constructor and finally the set of extended graphs is defined.

To express the the super relation we slightly extend the definition of type graph from the previous chapter.

**Definition 2.3.1** An *extended graph* $g$ is composed of the following five sets $(V, symb, args, sups, roots)$ where $(V, symb, args, roots)$ form a graph (without superclass relationship) and $sups : V \to V \bigcup \{NoSuper\}$ assignes to each node its super node. The set of nodes without super type is $Base_g = \{n \in V : symb(n) = Object\}$. Moreover the following must be satisfied:

(i) for each $n \in V$ the node $sups(n) \notin Var_g$, guarantees that variable cannot be super type and

(ii) $\{n : sups(n) = NoSuper\} = Base_g \bigcup Var_g$ ensures that exactly set of nodes representing *Object* and variables does not have super type.

As in chapter one we will work only with acyclic graphs. Because of the addition of the super relationship between nodes we have to slightly modify

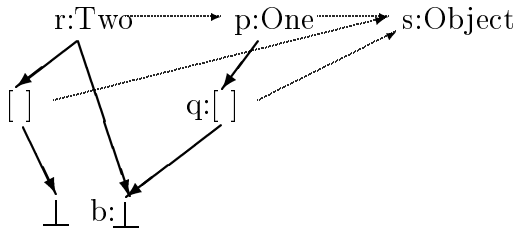


Figure 2.2: Directions on extended graphs

this term. This modification alters the meaning of many of the following terms whose definitions are based on direction. In this chapter we work only with extended graphs, so the change of definition should not be confusing.

**Definition 2.3.2** Let $g$ be extended graph and let $v \in V_g$. Then
(i) *Direction* is any element of $Dir = (\mathcal{N} \bigcup \{\mathcal{S}\})^*$,
(ii) *Path along direction* $s \in Dir$ is any $p \in {V_g}^*$ where $length(p) = length(s) + 1$ and for each $i < length(p)$ where

$$p_{i+1} = args_g(p_i)_{s_i} \quad \text{if } s_i \in \mathcal{N}$$
$$p_{i+1} = sups_g(p_i) \quad \text{if } s_i = \mathcal{S}$$

Terms mentioned in following paragraphs remain unchanged as in the previous chapter but use the new definition of direction. This includes *possible direction* from node $u$ defined as the direction for which there is a path along it starting from $u$. The definition of *set of all possible directions from a node* is unchanged, using the extended directions.

**Example 2.3.1** Type `Two [a] b` on figure 2.2 extends `One [b]` which extends `Object`. Possible directions from the root $r$ are $(\mathcal{S}, \mathcal{S})$, $(\mathcal{S}, 0, \mathcal{S})$, $(\mathcal{S}, 0, 0)$, $(0, \mathcal{S})$, $(0, 0)$, $(1)$ and the zero length direction.

The result of an *application of direction to a node* is a node such that the direction leads from the first node to it.

**Example 2.3.2** The set of *used nodes* now also includes super nodes. On figure 2.2 the $Uses_g(p) = \{b, q, s\}$.

The cyclic graph is still defined as the one that contains node $v$ for which there is a direction $s$ with $length(s) \geq 1$ such that $v \overset{s}{\rightsquigarrow}_g v$. The direction can use super edges as is shown at 2.3

Because the meaning of *used nodes* is a slightly different the way *subgraph* is constructed changes appropriately. A subgraph of graph ( 2.2 ) rooted at $p$ consists of nodes $\{p, b, q, s\}$ and edges between them.

Nearly all nodes have super node, and it can have another one, and so on. That is why we define a set that assignes all super nodes to each node. We use the directions composed only from $\mathcal{S}$ (super direction) for this purpose.

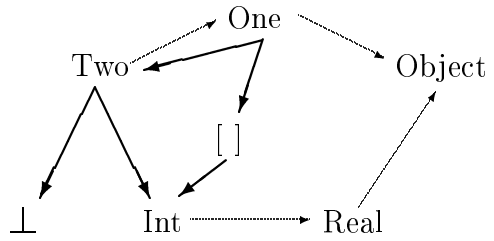


Figure 2.3: Cyclic extended graph

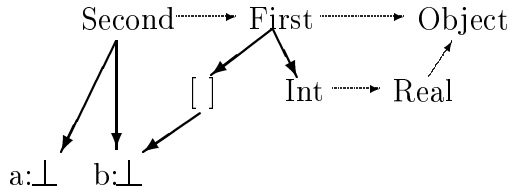


Figure 2.4: Constructor Second

**Definition 2.3.3** Let $g$ be a graph and $n \in V_g$. The set of *all super nodes* for a node $n$ is $supers_g(n) = \{u \in V_g : (\exists s \in \mathcal{S}^*)\ n \overset{s}{\rightsquigarrow}_g u\}$.

## 2.4 Constructors

In this section some existing object oriented languages are studied and some requirements for our type theory are stated. Then the definition of constructors from chapter one is extended to satisfy the requirements. The basic set of constructors is defined and fixed for the rest of this chapter.

The constructors in the previous chapter were represented by the given set $C$ and their arity. This can remain but to represent the hierarchy of graphs to each constructor (other then *Object* and $\perp$) the super type must be assigned.

In object oriented languages like Smalltalk and Java where each type has one super type, the type is fixed during runtime. This restriction is meaningful because by creation of a new subtype one reuses and modifies code of the super object. That is why we suppose that the super type for any constructor must remain the same too. During definition of own constructor, the programmer can specify its name, arity and super type. The super type can use only variables introduced by the constructor.

**Example 2.4.1** In figure 2.4 the constructor `Second` with arity two is defined. It establishes two variables `a` and `b` and extends type `First [b] Int`. In our extension to Clean records the constructor `Second` shown on picture 2.4 would be introduced by the code `Second a b = { First [b] Int & ... }`.


Figure 2.5: Graph for type Object

Another question that comes to mind is when we know that `A` is a super type of `B`, is `[A]` also super type of `[B]`? It seems reasonable to answer yes, because all operations on the list of `A` can also be performed on the list of `B`. But other constructors may behave in different ways. For example the function constructor `X -> Y` is super type of `A -> B` if and only if `Y` is super type of `B` and `A` is super type of `X`. So the constructor `->` is covariant in the former argument and contravariant in the latter.

```
// unused argument a
::Unused a = { Object & variable :: Int }
```

Algorithm 2.3: Unused argument of a type

```
// contravariant and covariant argument
::Restricted a = { Object & function :: a -> a }
```

Algorithm 2.4: Contravariant and covariant argument

There are also other possiblities. There can be argument that can change without any restriction and still create new subtypes (see 2.3 ) and also restricted parameter that cannot change (see 2.4 ).

**Definition 2.4.1** The set of constructors is specified by the immutable set $C$, arity function $arity : C \to \mathcal{N}$, function *hierarchy* that to each constructor assignes representing extended graph and function $variance : Var_g \to \{\nabla, \triangle, \Box, \Diamond\}^*$. So each argument has given variance kind.

Restriction to function *hierarchy* will be discused later. At present only $length(variance(c)) = arity(c)$ must hold for each $c \in C$. Each set of constructors will define a set of types that can be created from these constructors.

**Definition 2.4.2** Let $D \subseteq C$. The *set of extended graphs over these constructors* is denoted by $G_D$ and defined as the set $\{(V_g, symb_g, args_g, sups_g, roots) : (\exists f\ substitution)(\exists c \in D)\ g \equiv f(c) \wedge roots \in {V_g}^*\}$.

Now we give additional restriction to the set $C$. We sort all constructors and index them by integers. Let it be ordered by integers $C = \{\bot\} \bigcup \{c_0, c_1, \ldots\}$. Moreover $c_0 = Object$. Let $C_i = \{c_0, \ldots, c_i\}$. The $i+1$-th constuctor's subclass must be defined in the set of types created from the previous $i$ constructors.

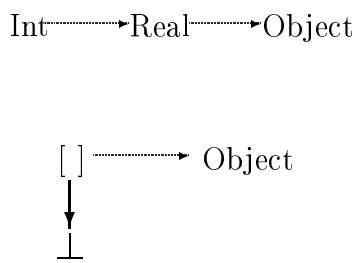


Figure 2.6: Standard types

**Definition 2.4.3** The definition of function *hierarchy* must satify:
(i) $hierarchy(Object)$ = graph in figure 2.5 with root $r$, formally speaking $(\{r\}, \{(r, Object)\}, \{(r, \lambda)\}, \langle r\rangle, \{(r, NoSuper)\})$,
(ii) $g \mid sups_g(r) \in G_{C_i}$, where $g = hierarchy(c_i + 1)$ is an acyclic extended graph. $r$ is its only root and $symb_g(r) = c_i$. The set of variables is restricted to $Var_g = \{args_g(r)_i : i < arity(c_i)\}$.

**Notation 2.4.1** The *set of graphs over all constructors* is denoted by $G = \bigcup_{i=0}^{\infty} G_i$, where $G_i$ is set of graphs over $\{c_0, \ldots, c_i\}$.

The step by step method of construction of $G$ reflects the process of introduction of new types in the program. It guarantees that no constuctor has itself as super type. Moreover it allow us to prove lemma showing that for graphs from $G$ we need not care about the super type when comparing two nodes for equivalency. It is sufficient to compare symbols and arguments.

**Lemma 2.4.2** Let $g \in G$, let $u, v \in V_g$ and $u, v \notin Var_g$. Then

$$u \equiv_g v \Leftrightarrow symb_g(u) = symb_g(v) \wedge (\forall i)\ args_g(u)_i \equiv_g args_g(v)_i$$

*Proof.* "$\Rightarrow$" Simplier case. As shown in lemma 1.8.2 from the previous chapter and because the equivalence here is an extended case of the equivalence used there (directions can reach super type also), two equivalent nodes must have same symbols and equivalent parameters.
"$\Leftarrow$" Because $symb_g(u) = c = symb_g(v)$ both graphs were created from the same shape for constructor $c$, the graph $g = hierarchy(c)$ with root $r$, by different substitutions. Each assigns the $i$-th constructor variable $i$-th argument of the node $(f_u(args_g(r)_i) = args_g(u)_i)$. Because for each $i$ the $args_g(u)_i \equiv_g args_g(v)_i$ also the resulting graphs are equivalent. □

For the following we suppose that the set of constructors include such types as `Int`, `Real` and `[]`. Their definitions are shown at figure 2.6 .

## 2.5 Comparing of Types

Two methods of comparing types are discussed in this chapter. One is based on the more general term introduced in chapter one. The second is specific to hierarchy types described in this chapter. As in the chapter one we define the more general term.

**Definition 2.5.1** We say that graph $g$ is *more general* then graph $h$ if there exists a substitution $f$ such that $f(g) \equiv h$.

**Example 2.5.1** When a actual parameter is applied to a function in the subtyping theory, it need not necessarilly match exactly type of a formal parameter. It can be its subtype. So the well-known example of function `last :: [a] -> a` for the following application would not lead to equality but inequality, $Int$ should be subtype of $a$.

```
last [1, 2, 3]
```

In general, any application leads to inequality. That is the reason for defining the subtype relation between types.

**Definition 2.5.2** Let $g \in G$. Let $u, v \in V_g$. The node $u$ is a subtype of $v$, that is denoted by $u \leq_g v$ if there exists a node $n \in supers_g(u)$ such that $symb_g(n) = symb_g(v)$ and for each $i$ the following is applied:

$$\begin{array}{lcl} variance(symb_g(n)) = \nabla & \Rightarrow & args_g(n)_i \leq_g args_g(v)_i \\ variance(symb_g(n)) = \triangle & \Rightarrow & args_g(v)_i \leq_g args_g(n)_i \\ variance(symb_g(n)) = \diamond & \Rightarrow & args_g(n)_i \leq_g args_g(v)_i \wedge args_g(v)_i \leq_g args_g(n)_i \\ variance(symb_g(n)) = \square & \Rightarrow & true \end{array}$$

The definition is recursive, because the result of comparison for nodes depends on their arguments. But the relation $\leq_g$ is defined only on acyclic graphs and that is the reason we can give the following proposition.

**Lemma 2.5.1** Let $g \in G$. Then the relation $\leq_g$ is reflexive, transitive and antisymetric.

*Proof.* The proof is based on induction on the number of used nodes of compared nodes.□

Program 2.5 is an implementation of comparison between nodes in an extended graph.

## 2.6 Node Inequalities

A set of inequalities on a graph $g$ is a set of node pairs compared either for type equality or subtype relationship. The solution is any graph $h$ less general then $g$, where nodes in pairs satisfies the conditions.

```
// Checks whether arg1 is subtype of arg2
checkSubType :: V_g V_g -> Bool;
checkSubType u v
| u == v = True;
| symb_g(u) == ⊥ = False;
| symb_g(u) == symb_g(v) = checkArguments
  (checkSubType, flip checkSubType, checkEquiv, \x->True) u v;
| symb_g(u) == Object = False;
| checkSubType sups_g(u) v

// Checks whether arg1 and arg2 are equivalent
// in terms of subtypes
checkEquiv :: V_g V_g -> Bool;
checkEquiv u v = checkSubType u v && checkSubType v u;

::TestFun :== V_g V_g -> Bool;
// Four functions for (variance, covariance, both, none) cases
::FourTestFuns :== (TestFun, TestFun, TestFun, TestFun);

// Takes four functions and two nodes and applies
// these functions to the nodes' arguments.
// The applied function is choosen on the
// knowledge of variance of the constructor.
// The result is true if all functions retuned true.
checkArguments :: FourTestFuns V_g V_g -> Bool;
checkArguments (f1, f2, f3, f4) u v
| symb_g(u) <> symb_g(v) = False
= all [
  case of (variance_i) {
    ∇ -> f1 args_g(u)_i args_g(v)_i;
    △ -> f2 args_g(u)_i args_g(v)_i;
    ◇ -> f3 args_g(u)_i args_g(v)_i;
    □ -> f4 args_g(u)_i args_g(v)_i;
  } \\ i <- [0..len - 1]
];
where {
  variance = variance(symb_g(u));
  len = arity(symb_g(u));
};
```

Algorithm 2.5: Algorithm for testing subtype relation between nodes

**Definition 2.6.1** Let $g$ be a graph. The *set of inequalities* is any pair $(g, M)$, where $M \subseteq \{u \preceq v : u, v \; are \; reachable\}$.

The reason for the restriction to only reachable nodes is the same as in the first chapter. We want equivalent graphs ($g \equiv h$) to behave similarly. However, since the equivalency of graphs depends only on reachable nodes, we have to restrict the inequalities to use only these nodes.

**Definition 2.6.2** Let $(g, M)$ be a set of inequalities. Let $f$ be a valid substitution on $g$. *Applying substitution* $f$ on $M$ can be denoted as $f(M)$ and declared as $f(M) = \{f(u) \preceq f(v) : u \preceq v \in M\}$.

The new set of inequalities $(f(g), f(M))$ is valid because all node pairs from $f(M)$ remain reachable.

**Definition 2.6.3** Let the pair $(g, M)$ be a set of inequalities. Then
(i) graph $g$ is the *solution* of $M$ if $(\forall \; (u \preceq v) \in M) \; u \leq_g v$,
(ii) The *set of solutions* of $(g, M)$ is $Sol_g(M) = \{h : (\exists f) \; f \; is \; a \; valid \; substitution \; on \; g \; \wedge \; h \equiv f(g) \wedge \; h \; is \; the \; solution \; of \; f(M)\}$,
(iii) graph $h$ is *the most general solution*, if for each solution $s \in Sol_g(M)$ it is true that $h \rhd s$.

The point (ii) guarantees that if a graph is a solution, then all equivalent graphs are also solutions. The transitivity of $\rhd$ allow us to prove a lemma similar to the one in the previous chapter. As a result, we obtain that if there is a solution, all less general graphs are also solutions.

**Lemma 2.6.1** Let $h_1 \in Sol_g(M)$ and $h_1 \rhd h_2$. Then $h_2 \in Sol_g(M)$.
*Proof.* As in chapter one. □

## 2.7 Solution for Set of Inequalities

This section discusses solutions of the set of inequalities. We give an example of set of inequalities that does not have most general solution and also present another set where the solution cannot be obtained only by applying substitutions. Lastly we present algorithm which is capable of deciding whether a given graph is a solution of the set of inequalities or not.

**Example 2.7.1** Figure 2.7 presents the graph created for the following expression:

```
// program without most general type
result :: Real;
result = func param;

param = 10;
```

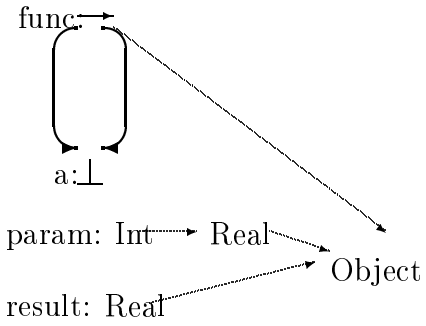


Figure 2.7: Graph for set of inequalities

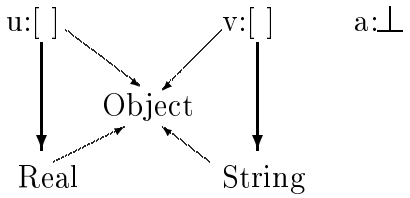


Figure 2.8: Missing common super type

The set of inequalities constructed for this piece of code includes $param \preceq a$ (because the integer parameter is applied to function with type `a -> a`) and $a \preceq result$ (the result of `func param` is assigned to `result`). There are two solutions for this set of inequalities. In both cases the type variable `a` must be something between `Int` and `Real`. Because there is no other type between these two, the `a` can be either `Int` or `Real`. But neither one of these results is more general than the other. So we have presented a simple set of inequalities that has a solution but does not have the most general one.

**Example 2.7.2** The model described in chapter one used only substitution to modify the graph while looking for a solution of a set of equalities. Conversely, if we use the same graph model for solving inequalities we could get into problems. Sometimes one needs to find a substitution for a variable bound by more subtypes and this can lead to searching for the greatest common super type that need not be present in the graph. An example is shown in figure 2.8 . Let $u \leq_g a$ and $v \leq_g a$. Then the substitution for `a` should be `[Object]`. But there is no node representing such a type in the graph. We have received type without node representation. This was impossible in the previous chapter and does not allow us to use the graphs as we used it before, as a base for computation with fixed set of nodes, where only edges are modified by substitutions.

```
// program that test whether the graph is a solution for
// the given set of inequalities
Test :: (Graph, Set of inequalities) -> Bool;
Test (_, ∅) = True;
Test (g, {u ≺ v ⋃ M}) = checkSubType u v && Test M;
```

Algorithm 2.6: Checking if graph is a solution of set of inequalities

So the requirements for solving sets of inequalities go beyond the model we have introduced in chapter one and that is why we present only an algorithm that is able to test whether a graph is a solution for set of inequalities. The program 2.6 checks whether all inequalities are satisfied.

We would like to check whether there is a solution for a given graph and set of inequalities. As in chapter one we defined a solution to be a substitution which produces a graph that satisfies all inequalities. But as we see in example 2.8 , sometimes the solution cannot be produced only by substitution. That is why looking for a set of solutions or only testing if there is a solution cannot be completed in the graph model we study. It requires additional operations.

## 2.8 Conclusion

Chapter two presents an extension to typing graphs that can express superclass relations between types with arguments. An extension to already existing Clean language is proposed which enriches the record types to objects.

We extend type graphs from the previous chapter to be able to work with objects. We modify all graph terms to reflect the change. Then we define what a valid constructor set is and how types are created from that set. We study real world requirements and we provide a way for comparing types for supertype relationship.

We introduce a set of inequalities and present an algorithm for testing whether a graph satisfies a set. Alas. We show examples where our model fails and is not suitable for the task. This is why we do not present an algorithm for finding most general solution (need not exist) nor an algorithm for testing if there is a solution for a set of inequalities (because there can be a solution not expressible in the terms of the graph model we are working with). The only algorithm we introduce is the one for checking if the set of inequalities is satisfied for a given graph.

# Conclusion

## Type graphs

In the previous chapters we have concentrated on presenting the typing theory in terms of graphs. We have defined a special kind of graph based on the graph presented in [3]. This graph contains nodes with assigned symbols, directed edges between them and a set of important nodes, roots. We introduced a special constructor that represented an empty node, variable.

We studied the original Hindley/Milner theory in the terms of our type graphs. We defined ways for comparing nodes in the graph and have shown that any set of equalities that has a solution can be solved by the algorithm presented . The algorithm not only finds a solution but finds the most general one. As a result we have shown that the Hindley/Milner theory can be presented by our type graphs.

The second task we concentrated on was to extend the theory to express subclassing of types. We defined our requirements based on the study of real object oriented languages extended by type variable polymorphism. We have shown how constructor and type sets should be constructed. After introducing the set of extended graphs we defined ways of comparing graph nodes and provided an algorithm that is capable of doing it.

We defined the set of inequalities, the solution and the most general solution of the set. In opposite to the chapter one we saw that even a set has solution it need not have the most general one. To fulfill the misery we gave an example showing that the graph model we used was not powerful enough to express the computation.

## Differences between theories

The success of the graph types in the case of Hindley/Milner theory and the failure in the extented case point out the differences. The theory presented in chapter one is based on substitutions. During the computation of the solution the graph is modified by substitutions but only to create a less general graph. The assigned type is never modified, only made more concrete.

The computation on the extended graphs is different. An inequality of variable cannot be solved only by substitution but rather by a range of subtypes.

The type variable has assigned a range of subtypes it can be substituted and each inequality can restrict the range to smaller one. Moreover the range is not only bounded by already existing types in the graph but can also be bound by new types not represented by nodes of the graph.

## Future research

The future research in the type graph area should focus on the range and new type problem. The solution to the range problem may lay in changing of variables to bound ones with the upper and lowest type that can be assigned to them. The problem of creation of new types could be solved by introduction of new operations (substitution is not enough) that would modify the type graph. But solutions to both problems remain open.

# Bibliography

[1] Barendregt, H. P. *The Lambda Calculus: its Syntax and Semantics*, North-Holland, Amsterdam, 1984.

[2] Rinus Plasmeijer, Marko van Eekelen *The Concurrent Clean Language Report*, http://www.cs.kun.nl/ clean/Clean.Cleanbook.html

[3] Barendregt, H. P., M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, M. R. Sleep, *Term Graph Rewriting*, http://www.cs.kun.nl/ clean/FP.Publications.html