Graph Rewriting Aspects of Functional Programming

Erik Barendsen Sjaak Smetsers
Computing Science Institute
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
e-mail erikb@cs.kun.nl, sjakie@cs.kun.nl

This work appeared in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1999, Chapter 2 (pp. 63–102).

Abstract

This chapter explains the relation between implementation of lazy functional programming languages and graph rewriting theory. We give an operational description of term graph rewriting in terms of elementary graph operations corresponding to memory allocation, pointer manipulation and garbage collection. Then we focus on three main topics: *computation* (evaluation to data values), *communication* (exchange of data) and *typing* (classification of objects and operations). Evaluation of function applications is formulated in the style of structural operational semantics. This gives rise to a natural notion of strictness. In the context of graph rewriting, communication involves copying of substructures. This is described in terms of homomorphisms and their relation to plain rewriting. Using a type inference system we characterize standard typing of graphs as well as a non-standard variant to analyze strictness properties.

1. Introduction

Graph rewrite systems (GRS) are generally considered as the most accurate foundation of modern functional languages and their implementations, see Plasmeijer and van Eekelen (1995) and Thompson (1996). Graph rewriting can be seen term rewriting extended with sharing. In term rewrite rules, multiple occurrences of variables lead to duplication of actual instances. In GRS's, such duplications are avoided by copying references to the objects rather than copying the objects themselves. Like in functional languages, in GRS's there is a separation between specifications (rewrite rules) and applications (graphs) of operations. Selection of an appropriate rewrite rule takes place via pattern matching.

Graph rewriting enables us to reason not only about the outcome of computation but also about the efficiency with which the result is obtained. A natural measure for efficiency with respect to time and space are the number of reduction steps and, respectively, the number of nodes needed during reduction.

In this chapter we will focus so-called *lazy* functional languages, in which redexes are reduced on demand, resulting in *lazy evaluation*. Lazy evaluation enables the specification of infinite data objects by means of finite expressions. This only works well if at any moment during program execution, only a finite initial segment of such a data object is needed, e.g. one should not try to count the number of elements of an infinite list. Besides infinite objects also infinite computations make sense, as long as those computations continuously lead to a 'gain of information'. An example of such an infinite computation is the execution of an operating system.

Rewrite rules specify possible transformations of graphs. There are several formal descriptions of graph transformations. The present paper deals with so-called *term graph rewriting* (Barendregt et al. (1987) and Barendsen and Smetsers (1994)). An equivalent formulation known as *equational graph rewriting* can be found in Ariola and Klop (1996). An alternative approach using category theory as a basis is given by Corradini et al. (1997).

In this chapter we will investigate how graph rewriting aspects play a role in (implementation of) functional programming languages. We will focus on *computations* (evaluation to data values), *communication* (exchange of data between processors) and *typing* (classification of objects and operations).

Traditionally, computations in rewrite systems are characterized by atomic rewrite steps and iteration of these via *reduction strategies*. Inspired by implementation practice, we describe evaluation of graph objects in an alternative way in the style of structural operational semantics. To this end we develop a graph syntax which enables us to characterize evaluation and typing in an elegant way.

Typing is a common feature of programming languages. In functional programming, standard type systems are used to ensure consistency of function applications during evaluation. These systems are usually based on first-order or polymorphic typed lambda calculus, see Barendregt (1992). We develop such a standard typing system for our graph syntax. This system can be extended in various ways to incorporate non-standard properties such as reference counting and normalization behaviour. In this chapter we will describe a typing system for analyzing termination behaviour of programs (so-called *strictness analysis*).

In a functional program, the evaluation of any expression is independent of the chosen reduction order due to the absence of side effects. This makes functional languages attractive for implementation on parallel hardware. Parallel computations involve communication between the separate processors. In the graph rewrite sense, exchange of data corresponds to copying part of a graph object. We will introduce copying as transformations of graphs and describe the interaction with ordinary rewriting.

The research leading to this paper was originally inspired by questions that arise from the implementation of the language *Clean*. This collaboration shows that combining theoretical research with development of an implementation can be very beneficial.

We have organized the chapter in such a way that each technical section is illustrated by an (informal) discussion connecting theory and practice. Section 2 and Section 3 introduce the basic aspects of graphs and graph rewriting respectively. In Section 4 graph rewriting is extended with copying for modelling data communication and data duplication in parallel implementations of functional languages. Lazy evaluation of graphs is described by structural operational semantics in Section 5. In Section 6, a strictness analysis method via typing is presented as an extension of standard polymorphic typing.

2. Term graphs

Term graph rewrite systems have been introduced in Barendregt et al. (1987), see also Barendsen and Smetsers (1994) and Plump (n.d.). This section summarizes some basic concepts.

The objects of our interest are directed graphs in which nodes are possibly labelled with a specific symbol. Nodes that are not labelled (the so-called *empty nodes*) have no outgoing edges. The number of outgoing edges of a non-empty node is determined by its symbol. One of the (empty or non-empty) nodes is indicated as the *root* of the graph. In the sequel we assume that $\mathcal{N}$ is some basic set of *nodes*, and Σ is a (possibly infinite) set of *symbols* with *arity* in $\mathbb{N}$. We use $\hookrightarrow$ to denote partial functions. If f is a partial function then $f(a)\downarrow$ expresses that f is defined on a; moreover $\uparrow$ denotes undefinedness.

2.1. DEFINITION. A *(rooted) term graph* (over Σ) is a quadruple

$$g = \langle N, symb, args, r\rangle$$

such that

(i) $N \subseteq \mathcal{N}$; N is the set of *nodes* of g;

(ii) $symb : N \hookrightarrow \Sigma$; *symb* is a partial function, $symb(n)$ is the *symbol* at node n;

(iii) $args : N \to N^*$ such that

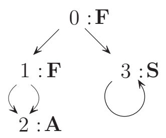
$$\begin{array}{rcll} \mathrm{length}(args(n)) & = & \mathrm{arity}(symb(n)) & \text{if } symb(n)\downarrow, \\ & = & 0 & \text{otherwise.} \end{array}$$

(iv) $r \in N$; r is the *root* of g.

Thus $args(n)$ specifies the outgoing edges of n. The i-th component of $args(n)$ is denoted by $args(n)_i$. A graph without empty nodes is called *closed*; otherwise it is said to be *open*.

We consider both finite and infinite graphs; for functional programming applications the finite graphs will be our basic objects.

2.2. Example. The following is a suggestive drawing of a rooted graph.



Here $N = \{0, 1, 2, 3\}$, $symb(0) = symb(1) = \mathbf{F}$, etc., arity$(\mathbf{F}) = 2$, arity$(\mathbf{S}) = 1$, arity$(\mathbf{A}) = 0$, $args(0) = (1, 3)$, $args(1) = (2, 2)$, etc., $r = 0$.

Some conventions: The elements of N are usually not explicitly indicated in the case of non-empty nodes; only the node symbols are displayed. Empty nodes are used to represent variables; we will suggestively use letters $(x, y, z, \ldots)$ to refer to these nodes.

2.3. Definition. Let $n \in g$. The *subgraph of* g *at* n (notation $g \mid n$) is the rooted graph $\langle N, symb, args, n\rangle$ where N is the set of nodes in g that are reachable from n, and $symb$ and $args$ are the restrictions (to N) of $symb_g$ and $args_g$ respectively.

2.4. Definition. Let g be a graph. An (*access*) *path* in g is a sequence

$$\pi = \langle i_1, i_2, \ldots, i_k\rangle$$

such that there exist nodes $n_1, n_2, \ldots, n_k$ (write $n_0 = r_g$) with

$$args_{i_j}(n_{j-1}) = n_j \quad \text{for } j = 1, \ldots, k.$$

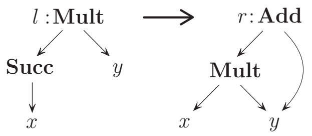
The node n_k is called the *destination* of π (notation $\mathrm{d}(\pi)$).

In Section 5 we will introduce an inductive syntax for graph expressions.

3. Graph rewriting

Rewrite rules specify transformations of graphs. A *rewrite rule* consists of a left-hand side (the *pattern*) and a right-hand side (the *result*). Both pattern and result are graphs. References from the result to pattern elements are indicated by variables, like in term rewriting.

For example, consider the following rule for multiplication.



We will often represent such rules in linear format:

$$\mathbf{Mult}(\mathbf{Succ}(x), y) \to \mathbf{Add}(\mathbf{Mult}(x, y), y).$$

This can also be interpreted as a term rewrite rule; note that in the graph interpretation the multiple occurrence of y in the result indicates sharing. Our format does not allow references from the result to non-variable pattern components. (This is not a serious restriction. For the general case the reader is referred to Barendsen and Smetsers (1994).)

In general, a rewrite rule R can be written as $F(\vec{p}) \to r$, where we use the vector denotation $\vec{p}$ as an abbreviation for $p_1, \ldots, p_n$. We have the usual restriction that the variables of the result r should occur in the pattern $F(\vec{p})$. If the result is a variable, then R is said to be *collapsing*, for example

$$\mathbf{Head}(\mathbf{Cons}(x, y)) \to x;$$

otherwise R is *extending*.

A graph g can be rewritten according to R if R is applicable to g, i.e., the pattern of R *matches* g. A *match* is a homomorphism from the pattern of R to a subgraph of g, i.e., a mapping that preserves the node structure.

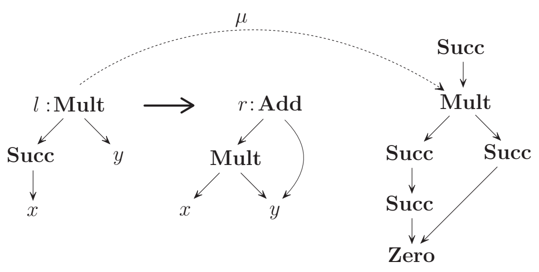
3.1. Definition. Let g, h be graphs, and $\varphi : N_g \to N_h$. Then φ is a *(graph) homomorphism* from g to h (notation $\varphi : g \to h$) if

$$\varphi(r_g) = r_h$$

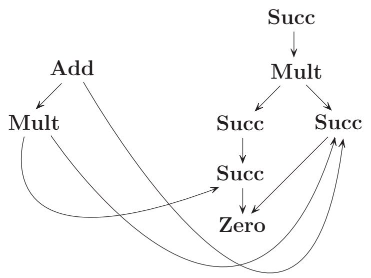
and moreover, for all $n \in g$

$$\begin{array}{rcll} \mathit{symb}_h(\varphi(n)) & = & \mathit{symb}_g(n) & \text{if } \mathit{symb}_g(n)\downarrow, \\ \mathit{args}_h(\varphi(n))_i & = & \varphi(\mathit{args}_g(n)_i) & \text{for } i = 1, \ldots, \text{arity}(\mathit{symb}_g(n)). \end{array}$$

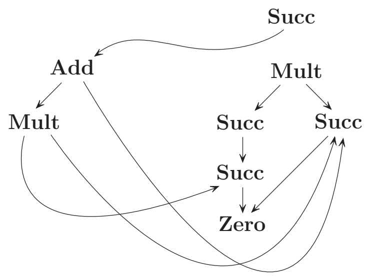
The combination of a rule and a match is called a *redex*. The following picture indicates a match μ for the above multiplication rule.



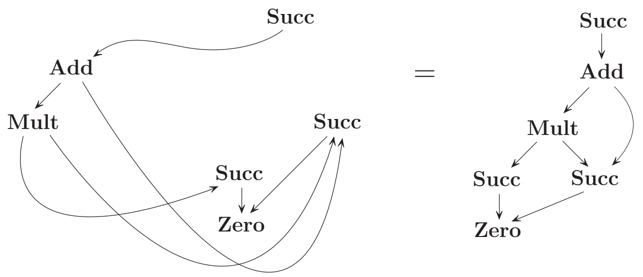
If a graph has a redex, it can be rewritten according to the structure of the right-hand side of the rule involved. This is done in three steps. Firstly, the graph is *extended* with (an instance of) the right-hand side of the rule. The connections from the new part with the original graph are determined by μ.



Then all references to the root of the redex are *redirected* to the root of the right-hand side.



Finally, all unreachable nodes are removed by performing *garbage collection*. Technically, this is equivalent to replacing the intermediate result g by its subgraph $g \mid r_g$.



3.2. Definition. A *graph rewrite system* (GRS) is a collection $\mathcal{R}$ of rewrite rules. Rewriting according to $\mathcal{R}$ is denoted by $\underset{\mathcal{R}}{\to}$.

Below we consider the notion of confluence, one of the basic properties of rewrite systems. Confluence is particularly important for programming applications of rewriting systems since it guarantees that results are independent of the way they are computed (*referential transparency*).

3.3. Definition. A GRS $\mathcal{R}$ is *confluent* (has the Church-Rosser property) if it satisfies the following diagram.

$$\begin{tikzcd} & \bullet \arrow[dl, two heads, "\mathcal{R}"'] \arrow[dr, two heads, "\mathcal{R}"] & \\ \bullet \arrow[dr, two heads, dashed, "\mathcal{R}"'] & & \bullet \arrow[dl, two heads, dashed, "\mathcal{R}"] \\ & \bullet & \end{tikzcd}$$

As in term rewriting, overlaps between patterns in rewrite rules are a source of non-confluency. To make this more explicit we introduce the following notions.

3.4. Definition. (i) Let g, g' be graphs. Then g and g' are *compatible* (notation $g \uparrow g'$) there exist a graph h and homomorphisms φ, φ' such that

$$\varphi : g \to h \text{ and } \varphi' : g' \to h$$

.

(ii) The rewrite rules $R_1 = F\vec{p} \to g, R_2 = G\vec{q} \to h$ are *overlapping* if for some non-empty n

$$F\vec{p} \mid n \uparrow G\vec{q}.$$

This overlap is called *trivial* if $R_1 = R_2$ and n is the root of $F\vec{p}$.

An alternative formulation of compatibility called *bisimilarity* appears in Ariola and Klop (1996) and Plump (n.d.).

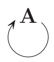
3.5. Definition. Let $\mathcal{R}$ be a GRS.

(i) $\mathcal{R}$ is *left-linear* if for each $R \in \mathcal{R}$ the pattern of R is a tree.

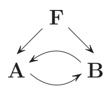
(ii) $\mathcal{R}$ is *non-overlapping* if there are no overlaps between rules in $\mathcal{R}$ that are non-trivial.

(iii) $\mathcal{R}$ is *orthogonal* if $\mathcal{R}$ is both left-linear and non-overlapping.

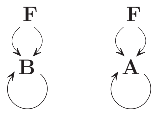
In contrast with term rewriting, orthogonal graph rewriting (on cyclic graphs) is not confluent as illustrated by the following system. Consider the two collapsing rewrite rules $\mathbf{A}(x) \to x$, $\mathbf{B}(x) \to x$, and note that a cyclic application of $\mathbf{A}$ (or $\mathbf{B}$)



reduces to itself (the rewrite step only consists of the trivial redirection of the $\mathbf{A}$-node to its argument). Now the graph



has two one-step reduction results:



Clearly, these results do not have a common reduct. This problem is caused by the existence of two so-called *interfering redexes*: redexes involving different collapsing rules, each having the root of the other redex as a result. A formalisation of the interference notion can be found in Barendsen and Smetsers (1994). Non interference can be obtained by admitting at most one collapsing rule or by prohibiting cycles. The absence of interference guarantees confluence, as stated in the following result.

3.6. THEOREM. *Let $\mathcal{R}$ be orthogonal and interference-free. Then $\mathcal{R}$ is confluent.*

An alternative approach to deal with interference is by extending the rewriting semantics in such a way that 'self-embedding redexes', like the cyclic **A**-application in the above example, reduce to $\bullet$ (a special graph constant called a *black hole*; see Ariola and Klop (1996)).

In functional programs we distinguish function and data symbols (called *constructors*), and usually consider rewrite rules in which patterns symbols, except the root, are data symbols. Such systems are called *function-constructor systems*.

3.7. CONVENTION. In the sequel we only consider function-constructor systems.

Graph rewriting in practice

Due to the presence of sharing, graph rewriting is a useful abstract model for describing implementations of functional languages. Functional programs can be thought of as rewrite rules and the rewrite semantics can be used to reason about the reduction behaviour of those programs. The execution time and memory consumption can be predicted by counting rewrite steps and graph nodes.

As an example, we give the following program that generates prime numbers. The algorithm is based in the well-known 'Sieve of Eratosthenes'. To increase readability, the operators $*$, $+$, $<$, $\neq$ are used as infix functions.

$$
\begin{array}{l}
\mathbf{Sieve}\ (g, prs) \rightarrow \\
\qquad \mathbf{IF}\ (\mathbf{IsPrime}\ (g, prs), \\
\qquad\quad \mathbf{Cons}\ (g, \mathbf{Sieve}\ (g+2, prs)), \\
\qquad\quad \mathbf{Sieve}\ (g+2, prs))
\end{array}
$$

$$
\begin{aligned}
&\textbf{IsPrime}\ (pr, \textbf{Cons}(f, r)) \rightarrow \\
&\qquad \textbf{Or}\ (f * f > pr, \textbf{And}\ (\textbf{Mod}(pr, f) \neq 0, \textbf{IsPrime}\ (pr, r)))
\end{aligned}
$$

Sieve's first argument is the next possible prime number. To decide whether this number is prime indeed, **Sieve** is also supplied with a list of previously generated prime numbers. If our programming languages allows for specification of typical graph constructions like cycles, one can use a cyclic list structure. This structure is generated by the following application of **Sieve** (the *node identifier* p is used to refer the the head of the result list).

$$
\begin{aligned}
&\textbf{Primes} \rightarrow \\
&\qquad p : (\textbf{Cons}\ (2, \textbf{Sieve}\ (3, p)))
\end{aligned}
$$

The cyclic structure enables **Sieve** to use its own output, avoiding the recomputation of prime numbers. This, however, results only in a small gain in efficiency; e.g. for computing the $30,000^{th}$ prime number, being $350,377$, **Sieve** only needs 118 prime numbers (the **IsPrime** predicate stops at the first prime above $\sqrt{350,377}$, which is 599). This example illustrates the high level of detail that can be achieved in reasoning about functional programs. In practice, this kind of reasoning is very useful for investigating functional programs with unexpected time and memory behaviour.

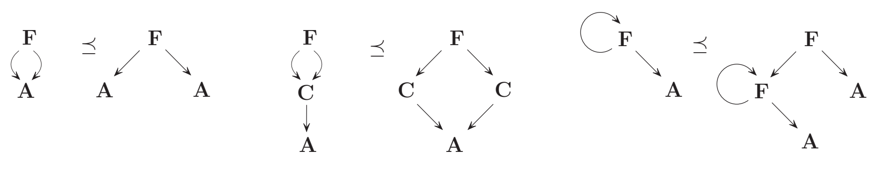
4. Copying

In this section we sill study the ordering $\preceq$ on closed graphs induced by graph homomorphisms.

4.1. DEFINITION. The *homomorphism ordering* $\preceq$ on closed graphs is defined as follows.

$$g \preceq h \quad \text{if} \quad \varphi : h \rightarrow g \text{ for some } \varphi.$$

For example.



Observe that if $g \preceq h$ the graph h results from *copying* part of graph g. This operation is sometimes called *unsharing* or *unravelling*.

Copying is interesting from a theoretical point of view, but has also significant implementation aspects. If one implements functional programming languages on parallel hardware separate processes communicate by exchanging data. Graph-theoretically this corresponds to copying part of a graph.

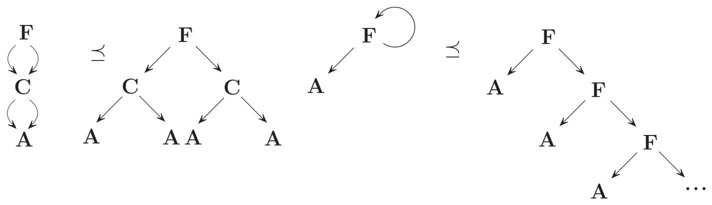
Copying also plays a role in sequential implementations. The use of certain resources of the underlying hardware (mostly for reasons of efficiency) makes it impossible to maintain sharing. For instance, if values are kept and manipulated in registers data is inevitably copied.

To each graph g a (possibly infinite) tree $\mathrm{T}(g)$ can be assigned, by completely *unravelling* g.

4.2. DEFINITION. Let g be a graph. The *tree unravelling* of g (notation $\mathrm{T}(g)$) is defined as follows. Write Acc_g for the collection of all access paths in g.Set $\mathrm{T}(g) = u$, where

$$\begin{aligned} N_u &= Acc_g, \\ symb_u(\pi) &= symb_g(\mathrm{d}(\pi)), \\ args_u(\pi)_i &= \pi * \langle i \rangle, \\ r_u &= \langle\,\rangle. \end{aligned}$$

E.g.

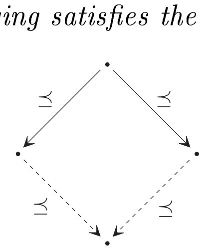


By $g \sim_{\mathrm{T}} g'$ we denote that g and g' are *tree equivalent*, i.e. have the same unravelling. (One can show that tree equivalence coincides with compatibility.) The following result is taken from Barendsen and Smetsers (1994).

4.3. THEOREM. *For each graph g the equivalence class of g with respect to $\sim_{\mathrm{T}}$ is a complete lattice with $\mathrm{T}(g)$ as maximal element.*

We now study the combination of rewriting and unravelling. To facilitate the description we denote unravelling as a rewrite step, i.e. $g \underset{\preceq}{\to} g'$ if $g \preceq g'$.

4.4. PROPOSITION. (i) *Copying satisfies the diamond property,*



In the case that the given graphs are finite, the common copy result can be chosen finite as well.

(ii) *If $\mathcal{R}$ is orthogonal and interference-free then*

$$\begin{tikzcd} g \arrow[r, "\preceq"'] \arrow[d, "\mathcal{R}"'] & g' \arrow[d, dashed, twoheadrightarrow, "\mathcal{R}"] \\ h \arrow[r, dashed, "\preceq"'] & h' \end{tikzcd}$$

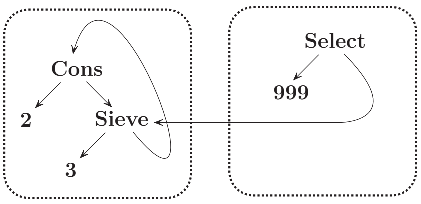
Unravelling might influence the redexes of a graph. The main result of this section states unravelling and ordinary reduction can be combined while preserving confluence.

4.5. THEOREM. *Let $\mathcal{R}$ be orthogonal and interference-free. Then $\mathcal{R}\preceq$-reduction is Church-Rosser.*

$$\begin{tikzcd} & \bullet \arrow[dl, twoheadrightarrow, "\mathcal{R}\preceq"'] \arrow[dr, twoheadrightarrow, "\mathcal{R}\preceq"] & \\ \bullet \arrow[dr, dotted, twoheadrightarrow, "\mathcal{R}\preceq"'] & & \bullet \arrow[dl, dotted, twoheadrightarrow, "\mathcal{R}\preceq"] \\ & \bullet & \end{tikzcd}$$

Copying in practice

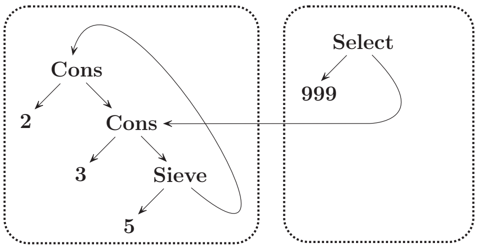
To illustrate that communication can be modelled via unravelling, consider the following snapshot of a computation of the 1000^{th} prime number (using the **Sieve** method of Section 3) on two processors. The idea is that any rewriting step in which pattern matching is involved can only take place if all data nodes of the pattern are available on the processor in question. Communication between processors consists of transmitting data, which corresponds to copying of constructor nodes.



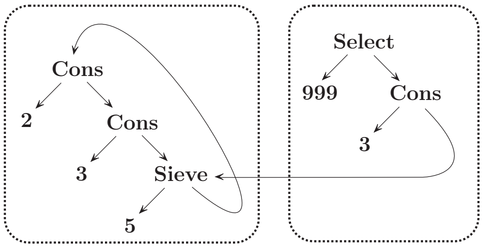
This first process generates prime numbers which are 'consumed' by the second process. The second process selects the 1000th element of the resulting list. The rules for select are

$$\begin{aligned} &\textbf{Select}\ (0, \textbf{Cons}(h,t)) \to h \\ &\textbf{Select}\ (n, \textbf{Cons}(h,t)) \to \textbf{Select}\ (\textbf{Dec}(n), t) \end{aligned}$$

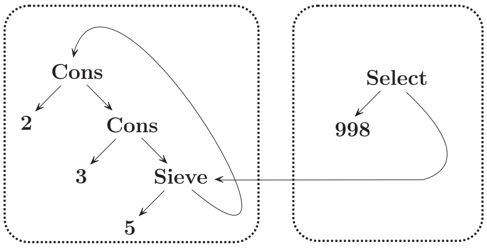
As soon as **Sieve** has computed the next prime, this number can be communicated.



This leads to a copy of both the **Cons**-node and the prime number itself. The computation of the further prime numbers remains shared. The result after copying is

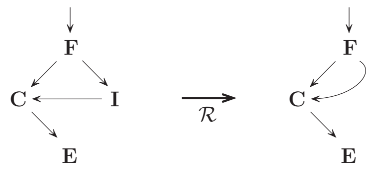


Now select has enough data to perform its next rewrite step.

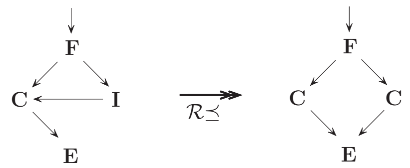


In general, the information transfer between processes is determined by the moment a processor needs data, and by the amount of information that is available (i.e. not subject to a present computation) at that moment. In terms of copying this means that it should be possible to defer copying at some points in the subject graph until these parts of the computation have been finished. For instance, in the example above copying is deferred at each **Sieve**-node until the next prime number has been computed. This way of copying is called *lazy copying* (see Barendsen and Smetsers (1994)). Lazy copying can be used to model communication *channels*, as proposed in van Eekelen et al. (1991). Together with a mechanism for creating parallel reduction processes, the lazy copying formalism provides a specification method for arbitrary process topologies with various kinds of communication links.

Copying also manifest itself in implementations of the redirection phase occurring at the end of each rewrite step. Instead of literally redirecting all references pointing to the function application to the root of the result, one can replace the symbol and arguments of the redex root by those of the contractum root. In case of a collapsing rule this optimisation leads to copying, e.g. the rewrite step (with $\mathbf{I}(x) \rightarrow x$)



is actually implemented as



5. Operational semantics

In this section we will elaborate on functional evaluation of graphs. Particularly, we will describe lazy evaluation in *Clean*.

Reduction strategies

In functional programming one is usually interested in data values as results of computations. In this section we will examine the graph rewriting equivalent of this process, i.e., reduction to constructors. From a traditional rewriting point of view, evaluation of objects is characterized in terms of so-called *reduction strategies*. Such a strategy is a function giving a particular rewrite result of an object (typically by pointing out one or more redexes to be reduced) if this object is not of the desired result shape (e.g. a normal form). Overall evaluation then takes place by iterating this function on the initial object. The so-called *left-most reduction strategy* in λ-calculus is a prime example. A typical research question is determining the normalisation properties of these strategies for particular rewrite systems.

When describing actual implementations of functional languages, this traditional approach is unsatisfactory. It abstracts from the way the redexes are determined, i.e. a strategy is considered as a *function* indicating redexes rather than a *process* searching for redexes.

In order to assess the efficiency of evaluation not only the number of reductions needed to compute the result is relevant. Also the complexity of strategy itself (i.e. the effort needed to determine which redexes to contract in each iteration) is important. For many theoretically interesting reduction strategies the searching process is far too complex to be of practical use. For this reason we will not use the modular approach isolating redex search from the iteration process, but describe evaluation as a whole. This will lead to a natural description of an evaluation procedure. Of course, this evaluation process can equivalently be characterized by a strategy in the original sense. However, the corresponding redex indicating function would be very hard to describe separately.

During the evaluation we will only be interested in an initial segment of the final data object, particularly its topmost constructor. Indeed, to determine whether a certain rewrite rule is applicable, it suffices to evaluate the function arguments up to the depth indicated by the pattern of that rule. Moreover, this lazy evaluation is the only way to handle infinite data. An object with a data constructor as root symbol is called a *constructor normal form* (or *cnf* for short). An evaluation method to cnf can easily be extended to a procedure for completely evaluating data objects (by recursively evaluating the arguments of the root of the cnf).

In the sequel we will describe a procedure (know as *functional evaluation*) for evaluating a graph object to cnf. The parameters of the procedure are the current graph in which a particular node is indicated as the *focus point*. The goal is to evaluate the subgraph at the focus point to cnf. This procedure is started with the focus point set to the root of the initial graph.

Given a graph and a current focus point, we are done if the focus point contains a data constructor. Otherwise the focus point refers to a function application. In that case we inspect the rewrite rules for the function in question and determine (according to the constructor patterns) which arguments need to be evaluated. For each of these patterns the evaluation procedure is recursively applied to the graph with the corresponding subgraph as focus point. Then the graph is rewritten according to the matching rewrite rule and we continue evaluation with the focus point at the contractum.

This is just a rough sketch. In fact, the choice of arguments to be evaluated in the recursive calls needs to be specified further. Since we focus on sequential evaluation we need to indicate the order in which the recursions occur.

The need to choose between various evaluation possibilities arises from the fact that a function may be defined by a collection of reduction rules, and that each rule pattern can have several non-variable arguments. We will describe one particular choice that has been employed in the implementation of *Clean*.

If a function application $\mathbf{F}(g_1, g_2)$ is evaluated then, apparently, one of the rules for $\mathbf{F}$ is applicable (after completion of the pattern). In *Clean* we consider one rule at the time and evaluate the arguments corresponding to the non-variable patterns parts of that rule from left to right. If, during evaluation, it becomes apparent that the current rule is not applicable (the pattern constructor differs from the root of the corresponding argument) one of the remaining rules is chosen and matching is continued. The choice of the next rule is determined by the order in which the rules are specified.

We illustrate this matching method by some examples. Consider the application $\mathbf{F}(g_1, g_2)$. If the rules for $\mathbf{F}$ are

$$\begin{array}{lll} \mathbf{F}(\mathbf{A}, x) & \rightarrow & \ldots \\ \mathbf{F}(x, y) & \rightarrow & \ldots \end{array}$$

then g_1 will be evaluated to cnf although the second rule is immediately applicable. Note that in *Clean* the above specification is interpreted as: 'If $\mathbf{F}$'s first argument is $\mathbf{A}$ then the first rule should be applied; in all other cases the second one.' This mechanism resembles the semantics of so-called *priority* rewriting systems, but the next example will show that it works differently. In case of the combination

$$\begin{array}{lll} \mathbf{F}(\mathbf{A}, \mathbf{B}) & \rightarrow & \ldots \\ \mathbf{F}(x, \mathbf{C}) & \rightarrow & \ldots \end{array}$$

we will evaluate g_1 and, subsequently, g_2. One might argue that the value of g_2 suffices to decide which rule is applicable. This shows the evaluation in *Clean* is not normalising in the traditional sense.

In the remainder of this section we describe functional evaluation in more detail. As an intermediate step we define a syntax for graph denotations. The next step is to formulate an operational semantics for these graph expressions. Due to the inductive structure of the expressions, this can be done in a convenient way. The operational semantics incorporates the movement of the focus point and the cnf-reduction at each indicated position. Finally, the *Clean* specific top-to-bottom, left-to-right treatment of rewrite rules and patterns is made explicit by translating graph rewrite rules into graph expressions.

Graph syntax

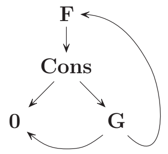
For a theoretical analysis of graph rewriting we need a proper representation of graph objects in which the essential structural aspects (such as application, sharing and cycles) are immediately visible. The plain 'structureless' formulation of Definition 2.1 is therefore not convenient, nor is the graph representation in linear form by means of systems of recursion equations. Instead, we will introduce a special *syntax* of *graph denotations*. An additional advantage is that it has an inductive structure, such that formulation of typing systems and a concise characterization of operational semantics can be done in an elegant way.

Pattern matching is expressed by an explicit construction $\mathsf{case} \cdots \mathsf{of} \cdots$. In function definitions, one can express pattern matching with respect to one argument at a time. This means that compound patterns are decomposed into nested ('sequentialized') case expressions fitting closely to the functional evaluation procedure.

Sharing of objects is expressed by a let construction. For cyclic sharing we use a recursion constructor μ. For example,

$$\mathsf{let}\ x = \mathbf{0}\ \mathsf{in}\ \mu z[\mathbf{F}(\mathbf{Cons}(x, \mathbf{G}(x, z)))]$$

stands for the graph



5.1. Definition. (i) The set of *graph expressions* is defined by the following syntax. Below, x ranges over variables, $\mathbf{C}$ over constructors and $\mathbf{F}$ over function symbols.

$$\begin{array}{lcl} E & ::= & x \mid \mathbf{C}(E_1, \ldots, E_k) \mid \mathbf{F}(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E\ \mathsf{in}\ E' \mid \\ & & \mu x[E] \mid \mathsf{case}\ E\ \mathsf{of}\ P_1|E_1 \ldots P_n|E_n \\ P & ::= & \mathbf{C}(x_1, \ldots, x_k) \end{array}$$

(ii) The set of free variables (in the obvious sense) of E is denoted by $\mathrm{FV}(E)$.

(iii) A *function definition* is an expression of the form

$$\mathbf{F}(x_1, \ldots, x_k) = E$$

where all the x_i's are disjoint and $\mathrm{FV}(E) \subseteq \{x_1, \ldots, x_k\}$.

We will not consider the degenerate cycle $\mu x[x]$ of length zero. Note that this would be the way to represent the 'black hole' $\bullet$ of Ariola and Klop (1996).

Each graph expression E can be translated into a graph $[\![E]\!]$, and function definitions into rewrite rules. For details see Barendsen and Smetsers (1996). The case constructions usually appear only at the outermost level of function definitions; these definitions are straightforwardly translated into rewrite rules with patterns for the function symbol in question. Other case expressions could be translated using auxiliary function symbols and rewrite rules; e.g.,

$$\begin{array}{lrcl} \mathsf{case}\ E\ \mathsf{of} & \mathbf{C_1}(x) & \mid & E_1 \\ & \mathbf{C_2}(y) & \mid & E_2 \end{array}$$

leads to the graph

$$\begin{array}{c} \mathbf{F} \\ \downarrow \\ [\![E]\!] \end{array}$$

with auxiliary rewrite rules

$$\begin{array}{rcl}\mathbf{F}(\mathbf{C_1}(x)) & \to & [\![E_1]\!],\\ \mathbf{F}(\mathbf{C_2}(y)) & \to & [\![E_2]\!].\end{array}$$

In the sequel we will assume that a fixed set of function definitions is given and use $\mathcal{R}$ to denote the translation of these into graph rewrite rules together with the necessary rules for the auxiliary functions.

Many syntactic operations have a graph-theoretic counterpart. We mention the relation between substitution and copying. Below, $[x := E]$ stands for substitution of E for all (free) occurrences of x.

5.2. PROPOSITION. $[\![\mathsf{let}\ x = E\ \mathsf{in}\ E']\!] \preceq [\![E'[x := E]]\!]$.

Operational semantics

We will describe evaluation of graph expressions in the style of *structural operational semantics*. The idea is to specify the result of a computation in a compositional, syntax-driven manner. This method focuses on the computation result (the so-called *value*) rather than on the computation process. Moreover, different evaluation orders (call-by-name, call-by-value) can be neatly expressed by varying specific clauses of the inductive specification, notably the rule for function application.

We will first describe a graph variant in which reductions are expressed by substitutions (*call-by-name*). With a small adjustment of the syntax we are able to express *call-by-reference*. We arrive at a system describing *lazy evaluation*, very similar to Launchbury (1993).

Copying semantics

We will specify evaluation to cnf. Recall that a graph is in cnf if its root node contains a data constructor. In graph expressions, such a constructor may appear as initial symbol, e.g.,

$$\mathbf{C}(E),$$

('simple constructor normal forms') but may also be 'hidden' in a sharing construction, e.g. in

$$\mathsf{let}\ x = E'\ \mathsf{in}\ \mathbf{C}(E)$$

and also in

$$\mu x[\mathbf{C}(x)].$$

We will first consider a system with plain cnf's as values. The other forms will be transformed into plain cnfs, thereby removing 'outermost sharing' by substitution. In the first example, E' is substituted for x in $\mathbf{C}(E)$. Note that this leads to copying, see 5.2. Cyclic objects will be unfolded one step. For a clear presentation we limit the set of constructors to $\square$, the binary constructor $\langle\ ,\ \rangle$ (intended meaning: pairing) and to the unary constructors inl and inr (intended meaning: injections of disjoint union). The $\square$ constructor is used as a representation of all elementary values; the other constructors represent compound values. To prepare for a typed variant, we assume that each case construct handles either the product constructor or the sum constructors, and not a combination of these.

It will be convenient to express the basic ideas and properties of the semantics in this system. Later we will present a more sophisticated system where sharing is preserved.

5.3. Definition. (i) An expression is in *constructor normal form* if it has one of the following shapes.

$$\square, \langle E, E'\rangle, \mathsf{inl}\ E, \mathsf{inr}\ E$$

(ii) Let E, V be expressions. Then E is said to *evaluate to* V (notation $E \Downarrow V$) if $E \Downarrow V$ can be produced in the following derivation system.

$$x \Downarrow x$$

$$\square \Downarrow \square \qquad \langle E, E'\rangle \Downarrow \langle E, E'\rangle$$

$$\mathsf{inl}\ E \Downarrow \mathsf{inl}\ E \qquad \mathsf{inr}\ E \Downarrow \mathsf{inr}\ E$$

$$\frac{E[\vec{x} := \vec{E}] \Downarrow V}{\mathbf{F}(\vec{E}) \Downarrow V} \quad \text{if } \mathbf{F}(\vec{x}) = E$$

$$\frac{E \Downarrow \mathbf{C}_j(\vec{E}) \qquad E_j[\vec{x} := \vec{E}] \Downarrow V}{\mathsf{case}\ E\ \mathsf{of}\ \ldots(\mathbf{C}_j(\vec{x}) \mid E_j)\ldots \Downarrow V}$$

$$\frac{E'[x := E] \Downarrow V}{\mathsf{let}\ x = E\ \mathsf{in}\ E' \Downarrow V} \qquad \frac{E[x := \mu x[E]] \Downarrow V}{\mu x[E] \Downarrow V}$$

We highlight some significant rules. Note that evaluating matching constructions like

$$\mathsf{case}\ E\ \mathsf{of}\ (\mathsf{inl}\ x|P)(\mathsf{inr}\ y|Q)$$

leads to evaluation of the subexpression E and subsequently (depending on the shape of the value of E) one of the two alternatives. Evaluation of a function application $\mathbf{F}(\vec{E})$ proceeds by substituting $\vec{E}$ for the formal parameters in the definition of $\mathbf{F}$.

The following result expresses the soundness of the semantics in terms of rewriting and copying.

5.4. PROPOSITION. *If $E \Downarrow V$, then*
(i) *V is in constructor normal form;*
(ii) $[\![E]\!] \xrightarrow[\mathcal{R}\preceq]{} [\![V]\!]$.

Sharing semantics

In order to keep track of sharing while decomposing expressions, we extend our graph representations with a list in which shared objects are stored.

5.5. DEFINITION. (i) An *(extended) graph representation* is a pair

$$G = \langle E \mid \rho \rangle$$

where E is an expression called the *main expression* of G, and ρ is a finite set of equations of the form $x = E'$ called the *graph store* of G.

(ii) An extended graph representation G is in *constructor normal form* if its main expression is a constructor normal form in the sense of Definition 5.3.

We identify graph representations $\langle E \mid \rho \rangle$ that only differ in the names of the bound variables (inside E or by ρ). The usual hygienic measures for avoiding name clashes between free and bound variables apply. The intention is that the representations $\langle E \mid x = E' \rangle$ and $\mathsf{let}\ x = E'\ \mathsf{in}\ E$ are equivalent, and so are $\langle E \mid \emptyset \rangle$ and E and the combination $\langle x \mid x = E \rangle$ and $\mu x[E]$. The interpretation of extended graph expressions as graphs is straightforward.

Now we can formulate evaluation of extended graph representations. For an easier presentation we require the topmost constructor of a value not to be shared. This will only make a difference if this topmost node is on a cycle; for example, $\langle x \mid x = \mathbf{C}(x) \rangle$ evaluates to its one-step unfolding $\langle \mathbf{C}(x) \mid x = \mathbf{C}(x) \rangle$. (Unsharing can be avoided at the cost of a more elaborate description of the semantics.)

5.6. Definition. Evaluation of extended graph representations is expressed by the following derivation system.

$$\langle \square \mid \rho\rangle \Downarrow \langle \square \mid \rho\rangle \qquad \langle\langle E, E'\rangle \mid \rho\rangle \Downarrow \langle\langle E, E'\rangle \mid \rho\rangle$$

$$\langle \mathsf{inl}\ E \mid \rho\rangle \Downarrow \langle \mathsf{inl}\ E \mid \rho\rangle \qquad \langle \mathsf{inr}\ E \mid \rho\rangle \Downarrow \langle \mathsf{inr}\ E \mid \rho\rangle$$

$$\frac{\langle E \mid \rho\rangle \Downarrow \langle V \mid \rho'\rangle}{\langle x \mid \rho, x = E\rangle \Downarrow \langle V \mid \rho', x = V\rangle}$$

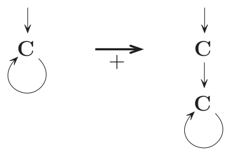
$$\frac{\langle E \mid \rho, \vec{x} = \vec{E}\rangle \Downarrow U}{\langle F(\vec{E}) \mid \rho\rangle \Downarrow U} \quad \text{if } \mathbf{F}(\vec{x}) = E$$

$$\frac{\langle E \mid \rho\rangle \Downarrow \langle \mathbf{C}_j(\vec{E}) \mid \rho'\rangle \qquad \langle E_j \mid \rho', \vec{x} = \vec{E}\rangle \Downarrow V}{\langle \mathsf{case}\ E\ \mathsf{of}\ \ldots(\mathbf{C}_j(\vec{x}) \mid E_j)\ldots \mid \rho\rangle \Downarrow V}$$

$$\frac{\langle E' \mid \rho, x = E\rangle \Downarrow U}{\langle \mathsf{let}\ x = E\ \mathsf{in}\ E' \mid \rho\rangle \Downarrow U} \qquad \frac{\langle E \mid \rho, x = E\rangle \Downarrow U}{\langle \mu x[E] \mid \rho\rangle \Downarrow U}$$

The first rule (for specifications with a variable as main expression) needs some explanation. Evaluating a variable x (with equation $x = E$ in the graph store) corresponds to evaluating a shared subgraph. This evaluation has a side-effect: not only is the value V of x given as result, but also updated in the graph store as $x = V$. This makes the result of evaluation available via the other references (occurrences of x) to the object. In the other rules, updating of graph stores during evaluation is taken into account by the transitions from ρ to ρ'. Note the uniform treatment of ordinary and cyclic sharing. In fact, there is no need to have two separate sharing constructs in the sharing semantics.

In order to express soundness we introduce the denotation $\underset{+}{\rightarrow}$ for the 'root unfolding' of cyclic graphs, e.g.,



Note that $g \underset{+}{\rightarrow} g$ if the root of g is not part of a cycle.

5.7. Proposition. *If* $G \Downarrow U$, *then*

(i) U *is in constructor normal form;*

(ii) $[\![G]\!] \underset{\mathcal{R}}{\twoheadrightarrow} \cdot \underset{+}{\rightarrow} [\![U]\!]$.

Strictness

We will introduce a notion of *strictness* for graph specifications and functions. Strictness information can be used for optimization of the evaluation process. For simplicity we formulate all notions in the framework of the copying operational semantics. These easily carry over to extended graph specifications.

5.8. DEFINITION. (i) Let E be an expression. E is *defined* (notation $E\Downarrow$) if $E \Downarrow V$ for some V. Otherwise E is undefined (notation $E\Uparrow$).

(ii) E is *strict in x* if for all expressions A

$$A\Uparrow \;\Rightarrow\; E[x := A]\Uparrow.$$

(iii) A unary function $\mathbf{F}$ (say with definition $\mathbf{F}(x) = E$) is *strict* if E is strict in x. (Likewise one formulates argumentwise strictness of functions of larger arity.)

Whenever a function $\mathbf{F}$ is strict, one could modify the evaluation of $\mathbf{F}$-applications by the call-by-value rule

$$\frac{E' \Downarrow V' \qquad E[x := V'] \Downarrow V}{\mathbf{F}(E) \Downarrow V} \quad \text{if } \mathbf{F}(x) = E$$

without affecting definedness of graph expressions. By evaluating function arguments before evaluating the function body one avoids duplicating unevaluated expressions in the case copying semantics. Even in case arguments are shared strictness information can be useful; see Section 6.

Translating rewrite systems

The last step in defining the semantics for graph rewriting is a description of the translation of rewrite rules into function definitions. The only aspect of functional evaluation that still needs to be expressed is the order in which rules and pattern arguments are evaluated. Remember that we only consider function-constructor systems.

Translation of unary functions is easy. For example,

$$\begin{array}{lll} \mathbf{Not}(\mathbf{True}) & \to & \mathbf{False} \\ \mathbf{Not}(x) & \to & \mathbf{True} \end{array}$$

is translated into the function definition

$$\mathbf{Not}(x) = \mathsf{case}\ x\ \mathsf{of}\ \begin{array}{l|l} \mathbf{True} & \mathbf{False} \\ \mathbf{False} & \mathbf{True} \end{array}$$

We will introduce some notational conventions.

First, we extend the case construction with an alternative that applies when none of the specified patterns matches. For example, the right-hand side of the above function can be written as

$$\mathsf{case}\ x\ \mathsf{of}\ \ \mathbf{True} \ \ | \ \ \mathbf{False} \\ \qquad\qquad\qquad\qquad | \ \ \mathbf{True}$$

Secondly, we give an alternative format for left-hand sides of rewrite rules in which each of the patterns is explicitly denoted by a variable. Each (possibly nested) pattern P then has following form.

$$P ::= x : \mathbf{C}(P_1, \ldots, P_k).$$

Furthermore, to be able to combine the different alternatives of a function, we separate the function symbol from the rest of the rewrite rule (i.e. pattern and right-hand side). This leads to the following 'matrix format' of function definitions.

$$F \quad \begin{array}{lcl} (P_{11}, \ldots, P_{1k}) & \rightarrow & g_1 \\ & \vdots & \\ (P_{l1}, \ldots, P_{lk}) & \rightarrow & g_l \end{array}$$

The translation of graph rewrite rules into functions definitions is given stepwise by the three transformation rules, called *Fun*, *Rul* and *Pat*. The construct **Fail** is used to represent an undefined value.

$$\mathit{Fun}(F \begin{array}{c} R_1 \\ \vdots \\ R_l \end{array}) = F(\vec{x}) = \mathit{Rul}(\langle R_1, \ldots, R_l \rangle, \vec{x}, \mathbf{Fail}) \quad \text{with all } \vec{x} \text{ fresh}$$

For each graph g, let E_g be a representation of g as a graph expression (see Barendsen and Smetsers (1996)). The function *Rul* is defined by induction on the length of the pattern sequence.

$$\begin{array}{lcl} \mathit{Rul}(\langle\rangle, \vec{x}, f) & = & f \\ \mathit{Rul}(\langle \vec{P} \rightarrow g, rs \rangle, \vec{x}, f) & = & \mathsf{let}\ \mathit{var}(\vec{P}) = \vec{x}\ \mathsf{in}\ \mathit{Pat}(\vec{P}, E_g, \mathit{Rul}(rs, \vec{x}, f)) \end{array}$$

Finally, for transforming patterns we define

$$\begin{array}{lcl} \mathit{Pat}(\langle\rangle, s, f) & = & s \\ \mathit{Pat}(\langle x, ps \rangle, s, f) & = & \mathit{Pat}(ps, s, f) \\ \mathit{Pat}(\langle x : \mathbf{C}\vec{P}, ps \rangle, s, f) & = & \mathsf{case}\ x\ \mathsf{of} \\ & & \quad \mathbf{C}\mathit{var}(\vec{P}) \ \ | \ \ \mathit{Pat}(\vec{P}, (\mathit{Pat}(ps, s, f), f) \\ & & \qquad\qquad\qquad | \ \ f \end{array}$$

Here $\mathit{var}\ (x : \mathbf{C}\vec{P}) = x$ and $\mathit{var}(\vec{P})$ is the obvious extension of *var* to sequences. To illustrate the procedure we give a concrete example: we translate the rules

$$\begin{array}{lcl} \mathbf{Mult}(\mathbf{Zero}, y) & \rightarrow & \mathbf{Zero} \\ \mathbf{Mult}(\mathbf{Succ}(\mathbf{Zero}), y) & \rightarrow & y \\ \mathbf{Mult}(\mathbf{Succ}(x), y) & \rightarrow & \mathbf{Add}(y, \mathbf{Mult}(x, y)) \end{array}$$

specified in matrix format as

$$\textbf{Mult}\quad \begin{array}{lcl} (x_1 : \textbf{Zero}, x_2) & \to & \textbf{Zero} \\ (x_3 : \textbf{Succ}(x_4 : \textbf{Zero}), x_5) & \to & y \\ (x_6 : \textbf{Succ}(x_7), x_8) & \to & \textbf{Add}(x_8, \textbf{Mult}(x_7, x_8)) \end{array}$$

The above procedure results in

$$\begin{array}{ll} \textbf{Mult}(z_1, z_2) = & \mathsf{let}\ (x_1, x_2) = (z_1, z_2)\ \mathsf{in} \\ & \mathsf{case}\ x_1\ \mathsf{of} \\ & \quad \textbf{Zero} \mid \textbf{Zero} \\ & \qquad\quad\ \mid \mathsf{let}\ (x_3, x_5) = (z_1, z_2)\ \mathsf{in} \\ & \qquad\qquad \mathsf{case}\ x_3\ \mathsf{of} \\ & \qquad\qquad\quad \textbf{Succ}(x_4) \mid \mathsf{case}\ x_4\ \mathsf{of} \\ & \qquad\qquad\qquad\qquad\quad \textbf{Zero} \mid x_5 \\ & \qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Mult}' \\ & \qquad\qquad\qquad\qquad \mid \mathit{Mult}' \end{array}$$

with

$$\begin{array}{ll} \mathit{Mult}' = & \mathsf{let}\ (x_6, x_8) = (z_1, z_2)\ \mathsf{in} \\ & \mathsf{case}\ x_6\ \mathsf{of} \\ & \quad \textbf{Succ}(x_7) \mid \textbf{Add}(x_8, \textbf{Mult}(x_7, x_8)) \\ & \qquad\qquad\quad \mid \textbf{Fail} \end{array}$$

As one can see, the proposed transformation introduces some redundant pattern matches and let expressions. A straightforward elimination of these leads to the following result.

$$\begin{array}{ll} \textbf{Mult}(x, y) = & \mathsf{case}\ x\ \mathsf{of} \\ & \quad \begin{array}{ll} \textbf{Zero} & \mid \textbf{Zero} \\ \textbf{Succ}(z_1) & \mid \mathsf{case}\ z_1\ \mathsf{of} \\ & \quad \begin{array}{ll} \textbf{Zero} & \mid y \\ \textbf{Succ}(z_2) & \mid \textbf{Add}(y, \textbf{Mult}(z_2, y)) \end{array} \end{array} \end{array}$$

6. Typing Systems

Typing systems in functional programming languages are used to ensure consistency of function applications: the type of each function argument should match some specified input type. In this section we will describe a *standard type system* for graph denotations and function specifications. Functions are regarded as *polymorphic*: their types are given in a schematic form. This standard type system can be extended to characterize properties such as reference counts and strictness. This is done by annotating the type expressions. A common aspect of these *non-standard type systems* is that they contain a refined analysis of *resources*, i.e. occurrences of input objects (the free variables in our graph denotations).

As a first step towards non-standard typing, we formulate standard typing in a *resource conscious* way, keeping track of multiple usage and discarding of input objects. Then we will describe a system for *strictness typing.* This system can be used to refine the notion of strictness (see Definition 5.8) and analyze strictness properties of functions.

Uniqueness typing deals with reference counting. It can be used to incorporate *I/O* and destructive updates in functional languages. For an exposition of uniqueness typing the reader is referred to Barendsen and Smetsers (1996). Resource conscious systems appear in logic to characterize substructural logics such as linear, affine and relevance logic.

Standard typing

We will restrict ourselves to a syntax with a limited number of constructors and matching expressions (as in section 5).

6.1. Definition. (i) The set of *standard graph types* is given by the following syntax. Below, α ranges over *type variables.*

$$\sigma \ ::= \ \alpha \mid (\sigma_1 \times \sigma_2) \mid (\sigma_1 + \sigma_2) \mid \mathbf{1}.$$

(ii) A *function type* (of arity k, say) is an expression of the form

$$(\sigma_1, \ldots, \sigma_k) \rhd \tau.$$

where $\vec{\sigma}, \tau$ are graph types. The types σ_i are called *argument types* and τ is the *result type.*

The function symbols are supplied with a type by a *function type environment* $\mathcal{F}$, containing declarations of the form

$$\mathbf{F} : (\sigma_1, \ldots, \sigma_k) \rhd \tau,$$

where k is the arity of $\mathbf{F}$. In this case we write

$$\mathcal{F} \vdash \mathbf{F} : \vec{\sigma} \rhd \tau.$$

6.2. Definition. Instantiation of function types is defined by the following rule ($[\alpha := \rho]$ denotes substitution).

$$\boxed{\dfrac{\mathcal{F} \vdash \mathbf{F} : \vec{\sigma} \rhd \tau}{\mathcal{F} \vdash \mathbf{F} : \vec{\sigma}[\alpha := \rho] \rhd \tau[\alpha := \rho]}}$$

For the sequel, fix a function type environment $\mathcal{F}$. Now we can develop a system of type assignment for expressions and function definitions.

6.3. DEFINITION. (i) A *basis* is a finite set of declarations of the form $x{:}\tau$ concerning distinct variables.

(ii) The type system deals with *typing statements* of the form

$$B \vdash E : \sigma,$$

where B is a basis. Such a statement is valid if it can be produced using the following derivation rules.

$$B \vdash \Box : \mathbf{1} \quad \text{(constant)}$$

$$B, x{:}\sigma \vdash x : \sigma \quad \text{(variable)}$$

$$\frac{\mathcal{F} \vdash \mathbf{F} : \vec{\sigma} \rhd \tau \qquad B \vdash E_i : \sigma_i}{B \vdash \mathbf{F}\vec{E} : \tau} \text{(application)}$$

$$\frac{B \vdash E : \sigma \qquad B \vdash E' : \tau}{B \vdash \langle E, E' \rangle : (\sigma \times \tau)} (\times\text{-I})$$

$$\frac{B \vdash E : (\sigma \times \tau) \qquad B, x{:}\sigma, y{:}\tau \vdash E' : \rho}{B \vdash \mathsf{case}\ E\ \mathsf{of}\ \langle x, y \rangle | E' : \rho} (\times\text{-E})$$

$$\frac{B \vdash E : \sigma}{B \vdash \mathsf{inl}\ E : (\sigma + \tau)} (+\text{-I-l}) \qquad \frac{B \vdash E : \tau}{B \vdash \mathsf{inr}\ E : (\sigma + \tau)} (+\text{-I-r})$$

$$\frac{B \vdash E : (\sigma + \tau) \qquad B, x{:}\sigma \vdash P : \rho \qquad B, y{:}\tau \vdash Q : \rho}{B \vdash \mathsf{case}\ E\ \mathsf{of}\ (\mathsf{inl}\ x|P)(\mathsf{inr}\ y|Q) : \rho} (+\text{-E})$$

$$\frac{B \vdash E : \sigma \qquad B, x{:}\sigma \vdash E' : \tau}{B \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau} \text{(sharing)}$$

$$\frac{B, x{:}\sigma \vdash E : \sigma}{B \vdash \mu x[E] : \sigma} \text{(cycle)}$$

(iii) The function environment $\mathcal{F}$ is *type correct* if each function definition is type correct, i.e. for $\mathbf{F}$ with type $(\sigma_1, \ldots, \sigma_k) \rhd \tau$ and definition $\mathbf{F}(x_1, \ldots, x_k) = E$ one has

$$x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k \vdash E : \tau.$$

The type system is *syntax directed*: every rule corresponds to one syntactic construction. This makes it possible to translate the rules into a *typing algorithm* for finding a *principal typing* for each expression, of which all other types for that expression can be obtained by substitution (see Barendsen and Smetsers (1995)). In the sequel we will focus on expression typing and tacitly assume type correctness of the function environment.

We will now give a resource conscious formulation of standard typing. In the rules for the specific syntactic constructions, the subexpressions are typed using disjoint sets of resources. The *weakening* and *contraction* rules deal with discarding resources and multiple occurrences (duplication) of resources.

6.4. DEFINITION. (i) The denotation B, B' stands for the *disjoint union* of bases, i.e. union of bases B, B' over disjoint sets of variables.

(ii) The relation $B \vdash E : \sigma$ is characterized by the following derivation system.

$$\vdash \square : \mathbf{1} \quad \text{(constant)}$$

$$x{:}\sigma \vdash x : \sigma \quad \text{(variable)}$$

$$\frac{\mathcal{F} \vdash \mathbf{F} : \vec{\sigma} \rhd \tau \qquad B_i \vdash E_i : \sigma_i}{\vec{B} \vdash \mathbf{F}\vec{E} : \tau} \text{(application)}$$

$$\frac{B \vdash E : \sigma \qquad B' \vdash E' : \tau}{B, B' \vdash \langle E, E' \rangle : (\sigma \times \tau)} (\times\text{-I})$$

$$\frac{B \vdash E : (\sigma \times \tau) \qquad B', x{:}\sigma, y{:}\tau \vdash E' : \rho}{B, B' \vdash \mathsf{case}\ E\ \mathsf{of}\ \langle x, y \rangle | E' : \rho} (\times\text{-E})$$

$$\frac{B \vdash E : \sigma}{B \vdash \mathsf{inl}\ E : (\sigma + \tau)} (+\text{-I-l}) \qquad \frac{B \vdash E : \tau}{B \vdash \mathsf{inr}\ E : (\sigma + \tau)} (+\text{-I-r})$$

$$\frac{B \vdash E : (\sigma + \tau) \qquad B', x{:}\sigma \vdash P : \rho \qquad B', y{:}\tau \vdash Q : \rho}{B, B' \vdash \mathsf{case}\ E\ \mathsf{of}\ (\mathsf{inl}\ x|P)(\mathsf{inr}\ y|Q) : \rho} (+\text{-E})$$

$$\frac{B \vdash E : \sigma \qquad B', x{:}\sigma \vdash E' : \tau}{B, B' \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau} \text{(sharing)}$$

$$\frac{B, x{:}\sigma \vdash E : \sigma}{B \vdash \mu x[E] : \sigma} \text{(cycle)}$$

$$\frac{B \vdash E : \tau}{B, x{:}\sigma \vdash E : \tau} \text{(weakening)}$$

$$\frac{B, x{:}\sigma, y{:}\sigma \vdash E : \tau}{B, z{:}\sigma \vdash E[x := z, y := z] : \tau} \text{(contraction)}$$

One easily shows that the two standard typing systems have the same set of derivable typing statements. In non-standard typing systems, the contraction rule is used to combine type information about two different occurrences of the same object.

Strictness analysis

There are several techniques for strictness analysis. We give a brief outline of the main ideas before focusing on one particular technique: strictness typing.

Strictness analysis by *abstract interpretation* introduces a non-standard semantics by translating functions into abstract versions over finite domains, notably over finite latices. The bottom elements of the domains play the role of generic 'undefined' values. Recursive abstract functions are defined by a fixed-point construction. The main property of this alternative interpretation is that it gives a decidable approximation of the (in general undecidable) strictness property, even in the higher-order case. This abstraction inevitably leads to loss of information. The standard form of abstract interpretation uses the two-point latice as ground domain. See Mycroft (1981), Burn et al. (1985) and Wadler (1987) for more information. The complexity of finding fixed points in abstract domains is one of the main reasons why abstract interpretation is unsuited for compilers for functional languages.

Abstract reduction analyses evaluation by mimicking reduction on sets of concrete values extended with generic elements for undefinedness. This technique approximates ordinary computations much closer than for instance abstract interpretation or strictness typing. The rewriting semantics is adjusted by specifying the behaviour of functions on non-standard elements. Abstract reduction sequences may not terminate. A special technique called *reduction path analysis* is used to cut off these sequences in a way that does keeps most of the strictness information intact; see Nöcker (1993).

Typing systems, treated below, are a purely syntactic ('intentional') way of deriving strictness information. The resulting strictness information only depends on the structure of the expressions, particularly on the occurrences of `case` -clauses, and (like in the case of abstract interpretation) not on the computational behaviour on concrete values. The advantage of strictness typing over abstract interpretation is that the first method can be combined with standard typing. Moreover one can adopt the Hindley-Milner solution for approximating fixed points. For more information, the reader is referred to Barendsen and Smetsers (1998) and Leung and Mishra (1991).

For strictness analysis via typing, standard typing will be annotated with a so-called *strictness attribute*. We use ! to indicate cnf strictness. The idea is to formulate strictness of E in x by a typing statement

$$x{:}\sigma^! \vdash E : \tau^!$$

('if x is undefined the E is undefined' or equivalently 'if E has a cnf then x has a cnf'). Strictness annotations are not restricted to the outermost level. For example,

$$p{:}(\sigma^! \times \tau)^! \vdash \mathsf{case}\ p\ \mathsf{of}\ \langle x, y\rangle | x : \sigma^!$$

expresses that if the case construction has a cnf, then p has a cnf (a pair) of which the first component has a cnf. This refines the notion of strictness. In the formal description it will be convenient to have an explicit denotation for the absence of strictness information. We will use ? for this purpose.

6.5. DEFINITION. (i) The collection of *strictness types* is obtained by annotating standard types. Below, u ranges over the attributes $\{!, ?\}$.

$$S \ ::= \ \alpha^u \mid (S_1 \times S_2)^u \mid (S_1 + S_2)^u \mid \mathbf{1}^u$$

(ii) The outermost attribute of S is denoted by $\ulcorner S \urcorner$.

(iii) We define an ordering on strictness types with the same underlying standard type via the ordering

$$! \leq ?$$

on attributes. E.g.

$$(\sigma^v \times \tau^w)^u \leq (\sigma^{v'} \times \tau^{w'})^{u'} \quad \text{iff} \quad u \leq u', v \leq v', w \leq w'.$$

(iv) Let S, S' be strictness types with the same underlying standard type. Then $S \sqcap S'$ denotes the type obtained by componentwise taking the minimum of the attributes in S and S'.

6.6. DEFINITION. Derivation of strictness typing statements $B \vdash E : S$ is characterized by the following rules.

$$\vdash \Box : \mathbf{1}^u \quad \text{(constant)}$$

$$x{:}S \vdash x : S \quad \text{(variable)}$$

$$\frac{\mathcal{F} \vdash \mathbf{F} : \vec{S} \rhd T \qquad B_i \vdash E_i : S_i}{\vec{B} \vdash \mathbf{F}\vec{E} : T}\,\text{(application)}$$

$$\frac{B \vdash E : S \qquad B' \vdash E' : T \qquad u \leq \ulcorner S \urcorner \qquad u \leq \ulcorner T \urcorner}{B, B' \vdash \langle E, E' \rangle : (S \times T)^u}\,(\times\text{-I})$$

$$\frac{B \vdash E : (S \times T)^u \qquad B', x{:}S, y{:}T \vdash E' : R \qquad \ulcorner R \urcorner \leq u}{B, B' \vdash \mathsf{case}\ E\ \mathsf{of}\ \langle x, y \rangle | E' : R}\,(\times\text{-E})$$

$$\frac{B \vdash E : S \qquad u \leq \ulcorner S \urcorner}{B \vdash \mathsf{inl}\ E : (S+T)^u}\,(+\text{-I-l}) \qquad \frac{B \vdash E : T \qquad u \leq \ulcorner T \urcorner}{B \vdash \mathsf{inr}\ E : (S+T)^u}\,(+\text{-I-r})$$

$$\frac{B \vdash E : (S+T)^u \qquad B', x{:}S \vdash P : R \qquad B', y{:}T \vdash Q : R \qquad \ulcorner R \urcorner \leq u}{B, B' \vdash \mathsf{case}\ E\ \mathsf{of}\ (\mathsf{inl}\ x | P)(\mathsf{inr}\ y | Q) : R}\,(+\text{-E})$$

$$\frac{B \vdash E : S \qquad B', x : S \vdash E' : T}{B, B' \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : T}\,\text{(sharing)} \qquad \frac{B, x : S \vdash E : S}{B \vdash \mu x[E] : S}\,\text{(cycle)}$$

$$\frac{B \vdash E : S \qquad T \leq S}{B \vdash E : T}\,\text{(subsumption)}$$

$$\frac{B \vdash E : S}{B, x{:}\tau^? \vdash E : S}\,\text{(weakening)}$$

$$\frac{B, x{:}S, y{:}T \vdash E : U}{B, z{:}S \sqcap T \vdash E[x := z, y := z] : U}\,\text{(contraction)}$$

In order to express the soundness of the system we refine the notion of definedness (see Definition 5.8 (i)) by parameterizing operational semantics by strictness types. For a clear presentation we restrict ourselves to the copying semantics.

6.7. DEFINITION. *Typed evaluation* is defined as follows.

(i) For each S, the notion $E \Downarrow_S V$ (E S-evaluates to V) is inductively defined by the following rules

$$\boxed{\begin{array}{c} E \Downarrow_{\sigma^?} E \\[1em] \Box \Downarrow_{\mathbf{1}^!} \Box \\[1em] \dfrac{E \Downarrow \langle E_1, E_2\rangle \qquad E_1 \Downarrow_S V_1 \qquad E_2 \Downarrow_T V_2}{E \Downarrow_{(S\times T)^!} \langle V_1, V_2\rangle} \\[1.5em] \dfrac{E \Downarrow \mathsf{inl}\ E' \qquad E' \Downarrow_S V}{E \Downarrow_{(S+T)^!} \mathsf{inl}\ V} \quad \dfrac{E \Downarrow \mathsf{inr}\ E' \qquad E' \Downarrow_T V}{E \Downarrow_{(S+T)^!} \mathsf{inr}\ V} \end{array}}$$

(ii) We say that E is S-*defined* if $E \Downarrow_S V$ for some V. Otherwise E is S-*undefined* (notation $E{\Uparrow_S}$).

Now soundness of strictness typing can be formulated. For one resource this can easily be expressed: if $x : S \vdash E : T$, then for all A one has

$$A{\Uparrow_S} \quad \Rightarrow \quad E[x := A]{\Uparrow_T},$$

cf. Definition 5.8.

For the general case we use an auxiliary notion involving substitution of undefined objects.

6.8. Definition. (i) A *graph environment* is a function ρ from variables to graph expressions. By $[\![E]\!]_\rho$ we denote the result of substituting $\rho(x)$ for x in E, for each free variable x of E.

(ii) We say that ρ *satisfies* $E : T$ (notation $\rho \vDash E : T$) if $[\![E]\!]_\rho{\Uparrow_T}$. Moreover, ρ satisfies a sequence $\vec{E} : \vec{T}$ if $\rho \vDash E_i : T_i$ for some i.

(iii) B *satisfies* $E : T$ (notation $B \vDash E : T$) if for all ρ

$$\rho \vDash B \quad \Rightarrow \quad \rho \vDash E : T.$$

6.9. Theorem. *For the strictness typing system we have*

$$B \vdash E : T \quad \Rightarrow \quad B \vDash E : T.$$

For details (including decidability issues) the reader is referred to Barendsen and Smetsers (1998).

As for the restricted notion of strictness, we can formulate call-by-value evaluation. For unary functions this becomes

$$\frac{E' \Downarrow_S V' \qquad E[x := V'] \Downarrow_T V}{\mathbf{F}(E) \Downarrow_T V} \quad \text{if } \mathbf{F}(x) = E \text{ and } \mathbf{F} : S \rhd T$$

Recursive data types

Thus far, our types admit only simple, non-inductive data objects. As an example of a system with a general recursion mechanism we will extend our treatment with *lists*.

Consider the following (recursive/inductive and polymorphic) type definition for lists

$$\mathsf{List}(\alpha) = \mathbf{Nil} \mid \mathbf{Cons}(\alpha, \mathsf{List}(\alpha)).$$

Apart from ordinary evaluation to cnf, there exist several useful kinds of evaluation of lists. All these evaluation methods use the cnf of a list (i.e. either a **Nil** node or a node with **Cons** a root constructor) as a starting point. Since **Nil** nodes are already in normal form, these nodes are not affected by further evaluation. In the case of a $\mathbf{Cons}(E_1, E_2)$ intermediate result we have the following possibilities.

- *head reduction* in which the argument E_1 is reduced to cnf;
- *spine reduction* in which (recursively) the argument E_2 is reduced, i.e. a list will be evaluated to expose its 'spine' $\mathbf{Cons}(E_1, \mathbf{Cons}(E_2, \ldots \mathbf{Nil} \cdot \cdot))$; here the elements E_i are not evaluated;
- *element reduction* which is a combination of head and spine reduction: the spine of a list is computed and all elements are reduced to cnf.

In our framework we indicate each evaluation kind by a specific strictness type. The intended evaluation is specified by extending the notion of definedness (Definition 6.7) with the following two rules.

$$\mathbf{Nil} \Downarrow_{(\mathsf{List}^u(S))^!} \mathbf{Nil}$$

$$\frac{E \Downarrow \mathbf{Cons}(E_1, E_2) \quad E_1 \Downarrow_S V_1 \quad E_2 \Downarrow_{(\mathsf{List}^u(S))^u} V_2}{E \Downarrow_{(\mathsf{List}^u(S))^!} \mathbf{Cons}(V_1, V_2)}$$

Now $(\mathsf{List}^?(\sigma^!))^!$ $(\mathsf{List}^!(\sigma^?))^!$ and $(\mathsf{List}^!(\sigma^!))^!$ correspond to head, spine and element evaluation respectively. Note that the evaluation rules are such that the spine constructors of the list (the **Cons** nodes) are treated uniformly. In general, such a uniform attribution of recursive occurrences of a type constructor leads to *homogeneous* objects: all 'recursive' parts of such an object are evaluated to the same degree. Without head reduction, the strictness types for lists actually represent Wadler's four point domain for denoting abstract list values, see Wadler (1987).

We will illustrate each evaluation kind by a characteristic unary function. The following function has a plain (cnf) strictness behaviour.

$$\begin{array}{lll}
\mathbf{IsEmpty} : (\mathsf{List}^?(\alpha^?))^! \rhd \mathsf{Bool} & & \\
\mathbf{IsEmpty}(x) = \mathsf{case}\ x\ \mathsf{of}\ \ \mathbf{Nil} & \mid & \mathbf{True} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{Cons}(x,y) & \mid & \mathbf{False}
\end{array}$$

The specified type of **IsEmpty** expresses that $\mathbf{IsEmpty}(\ell)$ is undefined whenever ℓ is undefined.

The second function illustrates head evaluation.

$$\begin{array}{l} \mathbf{Head} : (\mathsf{List}^{?}(\alpha^{!}))^{!} \rhd \alpha \\ \mathbf{Head}(x) = \mathsf{case}\ x\ \mathsf{of}\ \ \mathbf{Cons}(x,y) \ \ \mid \ \ x \end{array}$$

Spine evaluation occurs in

$$\begin{array}{lll} \mathbf{Length} : (\mathsf{List}^{!}(\alpha^{?}))^{!} \rhd \mathsf{Int} & & \\ \mathbf{Length}(x) = \mathsf{case}\ x\ \mathsf{of}\ \ \mathbf{Nil} & \mid & \mathbf{0} \\ \qquad\qquad\qquad\qquad\qquad\ \mathbf{Cons}(x,y) & \mid & \mathbf{Inc}(\mathbf{Length}(y)) \end{array}$$

$\mathbf{Length}(\ell)$ is only defined for lists ℓ of which the whole spine can be computed (in a finite number of steps). The elements of the list are not needed.

Element evaluation is needed in

$$\begin{array}{lll} \mathbf{Sum} : (\mathsf{List}^{!}(\alpha^{!}))^{!} \rhd \mathsf{Int} & & \\ \mathbf{Sum}(x) = \mathsf{case}\ x\ \mathsf{of}\ \ \mathbf{Nil} & \mid & \mathbf{0} \\ \qquad\qquad\qquad\qquad\quad \mathbf{Cons}(x,y) & \mid & \mathbf{Plus}(x, \mathbf{Sum}(y)) \end{array}$$

In an application $\mathbf{Sum}(\ell)$ not only the whole spine of ℓ is computed, but also each element (up to cnf).

Strictness analysis in practice

Essential for lazy evaluation is the occurrence of unevaluated objects. In an implementation one has to take care of these objects, and their evaluation whenever their values are needed. It turns out that this consumes large amounts of both space and time. This is mainly due to the fact that the computation of each value will require two representations during program execution, namely, the representation of the initial unevaluated expression (the so-called *closure*) and the representation of the final data object. Space consumption can be limited by avoiding double allocation of space via reuse of the space occupied by the initial expression when storing the final result. This does not reduce the time overhead: Filling the reserved space still takes place twice.

Compared to lazy evaluation, *eager evaluation* (or *call-by-value evaluation*) has the advantage that are no closures and therefore no overhead. Moreover, when dealing with data objects that have a fixed, statically known size (such as integers and reals) we can completely avoid the use of any storage by keeping such values on stacks or in registers (see Nöcker and Smetsers (1993)). This significantly increases the efficiency of programs in which these values occur.

We can conclude that in order to implement lazy evaluation efficiently, the creation of closures should be avoided as much as possible. Strictness analysis is a tool for achieving this: it indicates situations in which it is call-by-value evaluation is safe.

We will now explain how strictness information is actually used in an implementation. We first discuss how (fully) eager languages are usually implemented on conventional hardware.

Each rewrite rule is translated into a *subroutine*. Consider the following rule.

$$\mathbf{F}(x) \to \mathbf{G}(\mathbf{H}(x))$$

The generated subroutine will expect the input x either in a register or on a calling stack. The right-hand side of **F** constitutes the body of the subroutine: The (nested) application of **H** will result in a call to **H** (or, more precisely, to the subroutine corresponding to **H**); the result of **H** is directly passed to **G**. Finally **G** will be executed. A standard optimisation is to call **G** not via a subroutine jump but via an ordinary jump. This so-called *tail recursion elimination* prevents an unnecessary growth of the runtime calling stack. The technical reason is that there is no need to save the program counter as return address just before **G** is called.

Implementations of lazy languages cannot use the same translation scheme, since it is uncertain whether nested function applications in the right-hand side of a function definition (like $\mathbf{H}(x)$ in the above example) are really needed. In any case it is safe to evaluate the outermost application directly. Strictness information enables to indicate more 'strict applications'. For instance, if **G** is strict, then $\mathbf{H}(x)$ will be evaluated.

Using strictness information of functions, we can classify each node occurring in the right-hand side of a function definition either as *active* or as *passive*: The root of the right-hand side itself is active, and moreover, the arguments of an active function application are active as well if the corresponding function is strict in those arguments. All other nodes are passive. Active nodes can be evaluated immediately; passive nodes have to be built as graphs (as closures or as data objects). If a passive node is a closure, evaluation will take place as soon as its value is needed. In such a case the closure is prepared for evaluation (e.g., by moving actual arguments into registers) and the corresponding subroutine is called. The overhead due to this preparation can be significantly larger than the amount of time it takes to evaluate the function itself. This makes elimination of passive nodes worthwhile. In particular this occurs when the arguments and/or the result of such the involved function are basic values. In some cases, the use of strictness information can even lead to an efficiency gain of several orders of magnitude. For a more elaborate description of implementation aspects of lazy functional languages we refer to Smetsers et al. (1991).

7. References

References

Ariola, Z.M. and J.W. Klop (1996). Equational term graph rewriting, *Fundamenta Informaticae* **26**, pp. 207–240.

Barendregt, H.P. (1992). Lambda calculi with types, *in:* S. Abramsky, D.M. Gabbay and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press, pp. 117–309.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987). Term graph reduction, *in:* J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds.), *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE)* II, Eindhoven, The Netherlands, Lecture Notes in Computer Science 259, Springer-Verlag, Berlin, pp. 141–158.

Barendsen, E. and J.E.W. Smetsers (1994). Extending graph rewriting with copying, *in:* H.J. Schneider and H. Ehrig (eds.), *Graph Transformations in Computer Science, International Workshop*, Dagstuhl Castle, Germany, Lecture Notes in Computer Science 776, Springer-Verlag, Berlin, pp. 51–70.

Barendsen, E. and J.E.W. Smetsers (1995). Uniqueness type inference, *in:* M. Hermenegildo and S.D. Swierstra (eds.), *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, Utrecht, The Netherlands, Lecture Notes in Computer Science 982, Springer-Verlag, Berlin, pp. 189–206.

Barendsen, E. and J.E.W. Smetsers (1996). Uniqueness typing for functional languages with graph rewriting semantics, *Mathematical Structures in Computer Science* **6**, pp. 579–612.

Barendsen, E. and J.E.W. Smetsers (1998). Strictness typing. In preparation.

Burn, G.L., C.L. Hankin and S. Abramsky (1985). The theory of strictness analysis for higher order functions, *Proc. of Workshop on Programs as Data Objects*, DIKU, Denmark, Springer Verlag, LNCS 217, pp. 42–62.

Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe (1997). Algebraic approaches to graph transformation – Part I: Basic concepts and double pushout approach, *in:* G. Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. I, World Scientific, pp. 163–245.

van Eekelen, M.C.J.D., M.J. Plasmeijer and J.E.W. Smetsers (1991). Parallel graph rewriting on loosely coupled machine architectures, *in:* S. Kaplan and M. Okada (eds.), *Conditional and Typed Rewriting Systems (CTRS'90)*, Montreal, Canada, Lecture Notes in Computer Science 516, Springer-Verlag, Berlin, pp. 354–369.

Launchbury, J. (1993). A natural semantics for lazy evaluation,, *Proc. of POPL'93: Twentieth annual ACM symposium on Principles of Programming Languages*, Charleston, South Carolina, pp. 144–154.

Leung, A. and P. Mishra (1991). Reasoning about simple and exhaustive demand in higher-order lazy languages, *Proc. of International Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, Boston, USA, Springer Verlag, LNCS 523, pp. 328–351.

Mycroft, A (1981). *Abstract interpretation and optimising transformations for applicative programs*, Dissertation, University of Edinburgh.

Nöcker, E.G.J.M.H. (1993). Strictness analysis using abstract reduction, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, Kopenhagen, ACM Press, pp. 255–266.

Nöcker, E.G.J.M.H. and J.E.W. Smetsers (1993). Partially strict non-recursive data types, *Journal of Functional Programming* **3**, pp. 191–215.

Plasmeijer, M.J. and M.C.J.D. van Eekelen (1995). Concurrent Clean (version 1.3). Available via `http://www.cs.kun.nl/~clean/`.

Plump, D. (n.d.). Term graph rewriting. This Handbook, Chapter 1.

Smetsers, J.E.W., E.G.J.M.H. Nöcker, J.H.G. van Groningen and M.J. Plasmeijer (1991). Generating efficient code for lazy functional languages, *Proceedings of the Conference on Functional Languages and Computer Architectures (FPCA)*, Cambridge MA, USA, Lecture Notes in Computer Science 523, Springer-Verlag, Berlin, pp. 358–379.

Thompson, S. (1996). *Haskell: The Craft of Functional Programming*, Addison-Wesley, Wokingham.

Wadler, P. (1987). Strictness analysis over non-flat domains, *Abstract Interpretation of Declarative Languages*, Ellis Horwood.