Driving in the Jungle*

Jens Peter Secher

Department of Computer Science, University of Copenhagen (DIKU),
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.
Email jpscher@diku.dk

Abstract. Collapsed-jungle evaluation is an evaluation strategy for functional programs that can give super-linear speedups compared to conventional evaluation strategies such as call-by-need. However, the former strategy may incur administrative evaluation overhead. We demonstrate how this overhead can be eliminated by transforming the program using a variation of positive supercompilation in which the transformation strategy is based on collapsed-jungle evaluation. In penetrating the constant-factor barrier, we seem to be close to establishing a transformation technique that guarantees the efficiency of the transformed program. As a spin-off, we clarify the relationship between call-by-name, call-by-need and collapsed-jungle evaluation, showing that all three can be expressed as instances of a common semantics in which the variations — differing only in efficiency — are obtained by varying the degree of sharing in a DAG representation.

1 Introduction

Jungle evaluation has arisen from the graph grammar community as a means of speeding up evaluation of term rewrite systems, as described by Habel, Hoffmann, Kreowski and Plump[7,6]. In short, a jungle is a directed acyclic graph with explicit addresses of nodes. It has been shown that the naïve implementation of a function calculating Fibonacci numbers can be made to run in linear time by using evaluation on fully-collapsed jungles [7]. This kind of evaluation is achieved by never allocating new vertices if identical vertices are present in the graph.

The fully-collapsed-jungle approach has the drawback that it can be somewhat expensive to administer the graph. In this paper we will show how we can remove the run-time overhead of the above implementation technique by shifting the use of fully-collapsed jungles from run time to compile time. Specifically, we will do program transformation on fully-collapsed jungles instead of trees. The result is that we pay for the overhead once and for all, not every time a program is executed.

A spin-off of this approach is that we present a unified formalism for graph reduction, encompassing call-by-name, call-by-need, and collapsed-jungle evaluation (of which only the former has been omitted in this paper due to space

* Copyright © Springer-Verlag, to appear in Lecture Notes in Computer Science.
restrictions). We will show that these three reduction strategies can be captured in full by two abstract operations on graphs. We will do this by presenting a programming-language semantics parameterised over these two abstract operations; we will show that any implementation of these two abstract operations will result in the same semantics, as long as the implementation fulfills reasonable criteria. The distinction between any two implementations of the abstract operations is then only one of efficiency. This clarification seems interesting in its own right, since it makes it possible to compare variations of graph reduction.

2 Notation

We will use a notation that is somewhat non-standard, and an explanation is thus in order. When we write, say "a b c d", you should read this as ((a b) c) d. If a is a function, then a b c d means the result (if any) of ((a b) c) d. If a is an uninterpreted symbol (a constructor), then such an application means the term (i.e., tree) consisting of a root labelled a and ordered children b, c, and d.

We let \{a b c d\} denote the set containing the objects a, b, c, and d; we let \(a b c d\) denote the tuple containing these four objects; and we let \([a b c d]\) denote the list containing these objects. We use parentheses only as meta-syntax to group objects, that is, to avoid ambiguous interpretations. Thus \{(a b c d)\} denotes a singleton set (the element being whatever a b c d means). We use \(\cup\), \(+\), and \(\setminus\) as infix operators on sets to denote union, disjoint union, and subtraction, respectively. Given a set \(S\), the power set of \(S\) is denoted \(\mathcal{P}(S)\); the set of finite lists of elements from \(S\) is denoted \(S^*\).

We will often need to write sequences such as \(x_1 x_2 x_3 x_4 x_5\), and we will therefore introduce the shorthand notation \((x)_5^5\) for such a sequence. The superscript \("5^n"\) denotes that the preceding syntactic object "\(x\)" should be replicated five times, with the dot replaced by the consecutive numbers 1, 2, 3, 4, 5. If the replicated object is syntactically simple, we will leave out the dot altogether, and, for example, write \(x^n\) instead of \((x)_n^5\). When this kind of notation is used in several layers, the innermost part is expanded first. Hence \(\{(x \rightarrow t^2)^n\}\) means \(\{(x_1 \rightarrow t_1 t_2) \cdots (x_{n-1} \rightarrow t_1 t_2) (x_n \rightarrow t_1 t_2)\}\); the empty sequence is also allowed, so \(\{(x \rightarrow t)^0\}\) means \(\{\} = \emptyset\).

For a relation \(\rightarrow \subseteq S \times T\), we define the domain \(\mathcal{D}(\rightarrow) \triangleq \{s \mid \exists t : s \rightarrow t\}\). We say that \(\rightarrow\) is deterministic if, for all \(s \in S\), \(s \rightarrow t\) and \(s \rightarrow t'\) imply \(t = t'\), that is, \(\rightarrow\) is a (partial) function. To denote that \(f\) is a (partial) function, we write \(f \in S \rightarrow T\). If the domain of \(f\) is finite, we will use \(\{(s \rightarrow t)^n\}\) to denote the set of bindings that \(f\) comprises. If \(\rightarrow\) is a binary relation \(S \times S\), we denote by \(\rightarrow^+\) the transitive closure of \(\rightarrow\), and by \(\rightarrow^\ast\) the reflexive closure of \(\rightarrow^+\). The normal forms of \(\rightarrow\) is the set \(\mathcal{E}(\rightarrow) \triangleq S \setminus \mathcal{D}(\rightarrow)\).

3 Subject Language

Our programming language is a small, non-strict, first-order, function-oriented language with structured data and pattern matching.
Example 1 (Fibonacci Numbers). The following program defines the well-known Fibonacci function:

\[
\begin{align*}
\text{data} \ & \text{Nat} = 0 \mid s \text{Nat} \\
& \quad \text{fib} \ 0 \ = \ s \ 0 \ \\
& \quad \text{fib} \ (s \ x) \ = \ aux \ x \\
& \quad \text{aux} \ 0 \ = \ s \ 0 \\
\quad \text{aux} \ (s \ y) \ = \ \text{add} \ (\text{aux} \ y) \ (\text{fib} \ y) \\
& \quad \text{add} \ 0 \ y \ = \ y \\
& \quad \text{add} \ (s \ x) \ y \ = \ s \ (\text{add} \ x \ y)
\end{align*}
\]

Remark 2. The above program contains a data-type definition. For clarity, we will put such data-type definitions in our example programs, even though such data-type definitions are not permitted in the language.

3.1 Syntax

Definition 3. Assume denumerable disjoint sets of symbols for constructors \( C \), functions \( F \), pattern functions \( G \) (ranged over by \( c, f \), and \( g \), respectively), and variables \( \mathcal{X} \) (ranged over by \( x, y \)). Then the set of programs \( Q \), definitions \( D \), terms \( T \), and patterns \( P \) (ranged over by \( q, d, t, \) and \( p \), respectively) are defined by the abstract syntax grammar

\[
\begin{align*}
& \quad \text{(program)} \quad Q \ni q ::= d^m \\
& \quad \text{(definition)} \quad D \ni d ::= f \ x^n = t \mid (g \ p \ x^n = t.)^m \\
& \quad \text{(pattern)} \quad P \ni p ::= c \ x^n \\
& \quad \text{(term)} \quad T \ni t ::= x \mid c \ t^n \mid f \ t^n \mid g \ t^n \\
& \quad \text{(value)} \quad V \ni v ::= c \ v^n
\end{align*}
\]

where \( n \geq 0 \) and \( m > 0 \). We require that

1. No (pattern) function name is defined more than once.
2. No two patterns in a matcher definition contain the same constructor.
3. No variable occurs more than once in the left-hand side of a function definition (the definition is left-linear).
4. All variables in the body of a function definition are present among the variables in the left-hand side of the definition.

We let \( f \ x^n \overset{q}{=} t \) denote that the program \( q \) contains a definition \( f \ x^n = t \), and similarly for \( g \ p \ x^n \overset{q}{=} t \). As a shorthand, we let \( \mathcal{E} = C \cup F \cup G \). The set of variables in a term \( t \) is denoted \( \mathcal{V} \ (t) \), a term \( t \) is a ground term if \( \mathcal{V} \ (t) = \emptyset \).

We will shortly define the meaning of our little language in terms of a (parameterised) small-step operational semantics. The idea is that, given a ground term \( t \), we can determine which ground term \( t' \) (if any) that \( t \) reduces to in one step. The usual way to express such an reduction step is to define a relation on terms via substitutions.

Definition 4 (Substitution, renaming). Given a function \( \psi = \{(x. \rightarrow t.)^n\} \in \mathcal{X} \rightarrow \mathcal{T} \) from variables to terms, we denote by \( \theta = \{(x. \rightarrow t.)^n\} \in \mathcal{T} \rightarrow \mathcal{T} \) the substitution induced by \( \psi \).
3.2 Graphs

In this section we will investigate the interaction between various alternative representations of terms. We therefore formally introduce a more general representation of terms, namely directed acyclic graphs (DAGs).

**Definition 5 (Graphs).** Let $s$ range over a finite set of symbols $S$, and let $\alpha$ and $\beta$ range over a set of *addresses* $\mathcal{A}$. We then define the following:

1. We denote by *nodes over* $S$ the set $\mathcal{N}(S) \triangleq S \times \mathcal{A}^*$, ranged over by $\nu$. We will use the shorter notation $\nu = s \alpha^n$ instead of writing $\nu = \langle s [\alpha^n] \rangle$.
2. We denote by *directed graphs over* $S$ the set $\mathcal{A} \rightarrow (\mathcal{N}(S) + \mathcal{A})$ of partial mappings from addresses to nodes/addresses. Any directed graph $G$ induces a binary relation $\rightarrow \subseteq \mathcal{A} \times \mathcal{A}$ defined by $\beta \rightarrow \alpha_i$ iff $G \beta = \alpha_i$ or $G \beta = s \alpha^n \land i \leq n$.
3. We denote by *acyclic directed graphs over* $S$ the set $\mathcal{D}(S) \subseteq \mathcal{A} \rightarrow (\mathcal{N}(S) + \mathcal{A})$ of acyclic graphs (i.e., $G$ acyclic iff $\forall \alpha \in \mathcal{D}(G) : \alpha \nrightarrow \alpha$). We let $\triangledown$ range over the set of acyclic graphs.
4. We denote by *configurations over* $S$ the set $\mathcal{K}(S) \triangleq \mathcal{D}(S) \times \mathcal{A}$, ranged over by $\kappa$.
5. If an address is mapped to another address (i.e., not to a node), we call both the former address and the mapping an *indirection*, and we define

$$\|\triangledown\| \triangleq \{ \alpha \rightarrow \beta \mid \triangledown \alpha = \beta \in \mathcal{A} \}.$$  

The relation $\rightsquigarrow \subseteq \mathcal{K}(\mathcal{E}) \times \mathcal{A}$ is defined inductively by

$$\langle \triangledown \alpha \rangle \rightsquigarrow \alpha \quad \text{if} \quad \alpha \notin \mathcal{D}(\|\triangledown\|)$$

$$\langle \triangledown \alpha \rangle \rightsquigarrow \alpha' \quad \text{if} \quad \|\triangledown\| \alpha = \alpha'' \land \langle \triangledown \alpha'' \rangle \rightsquigarrow \alpha' .$$  

6. Let $\mathcal{A}$ be a set of addresses. We let $\triangledown \mathcal{A}$ denote the graph $\triangledown$ restricted to $\mathcal{D}(\triangledown) \setminus \mathcal{A}$.

7. We let fresh be a procedure that provides us with a *completely* new address each time it is called. We implicitly assume that every address mentioned has been drawn by this procedure; thus fresh will provide addresses that cannot be captured.

**Example 6 (DAG).** Let $\{0 \text{ add } \text{ fib aux}\}$ be a set of symbols and $\{\alpha \gamma \delta \beta \epsilon\}$ a set of addresses, and consider the graph

$$\triangledown = \left\{ \begin{array}{l}
\alpha \rightarrow \text{add} \quad \gamma \delta \\
\gamma \rightarrow \text{aux} \quad \epsilon \\
\delta \rightarrow \text{fib} \quad \epsilon \\
\beta \rightarrow \delta 
\end{array} \right\} .$$

Then $\mathcal{D}(\triangledown) = \{\alpha \gamma \delta \beta\}$, $\|\triangledown\| = \{\beta \rightarrow \delta\}$, $\langle \triangledown \beta \rangle \rightsquigarrow \delta$ and e.g., $\langle \triangledown \epsilon \rangle \rightsquigarrow \epsilon$.  

$\Box$
From the above example, you can see that we intend to use the set $\mathcal{E} = \mathcal{C} \cup \mathcal{F} \cup \mathcal{G}$ as the set of symbols that our graphs are defined over. We will now clarify the correspondence between such graphs and the terms of our little language. It is fairly straightforward to extract a term from a graph:

**Definition 7 (Extract).** Let $\phi \in \mathcal{A} \leftrightarrow \mathcal{X}$ be a unique bijection from addresses to variables. By $\phi_\alpha$ we denote the variable $\phi \alpha$. The function $\text{xtract} \in \mathcal{X}(\mathcal{E}) \rightarrow \mathcal{T}$ is defined inductively by

\[
\begin{align*}
\text{xtract} \ (\forall \alpha) & \triangleq \phi_\alpha, \text{ if } \alpha \notin \mathcal{D}(\forall) \\
\text{xtract} \ (\forall + \{ \alpha \rightarrow \beta \}) \alpha & \triangleq \text{xtract} \ (\forall \beta) \\
\text{xtract} \ ((\forall + \{ \alpha \rightarrow s \cdot \alpha^n \}) \alpha) & \triangleq s \ (\text{xtract} \ (\forall \beta))^n
\end{align*}
\]

**Example 8 (Extraction).** Assume $\phi_\alpha = x$, and consider the graph $\forall$ from Ex. 6; we have that $\text{xtract} \ (\forall \beta) = \text{fib} \ x$ and $\text{xtract} \ (\forall \alpha) = \text{add} \ (\text{aux} \ x) \ (\text{fib} \ x)$.

The opposite translation — from terms to graphs — however, depends on how much sharing of sub-terms we want to have. Furthermore, the precise formulation of operations on the graph representation of a term will depend on how far we are willing to go to maintain sharing of sub-terms. To abstract away from such preferences, we define two basic operations on our graphs. The first operation, $\text{upd}$, has the signature $\text{upd} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{X}(\mathcal{E}) \rightarrow \mathcal{G}(\mathcal{E})$. It takes as arguments a graph $\forall$, a target address $\alpha$, and a node $\nu$; Intuitively, calling $\text{upd}$ is like placing a piece in a jigsaw puzzle: the pockets in the new piece are attached to the tabs of the surrounding puzzle, and likewise the tabs of the piece are fitted into the surrounding pockets; more concretely, $\text{upd}$ updates $\forall$ with $\nu$ at $\alpha$ such that it connects with existing nodes in $\forall$. Formally, it must hold that

\[
\forall' = \text{upd} \forall \alpha \ (s \cdot \alpha^n)
\]

implies

\[
\forall \beta \in \mathcal{A} : \text{xtract} \ (\forall' \beta) = (\{ \phi_\alpha \rightarrow s \ (\text{xtract} \ (\forall \alpha))^{\beta} \}) \ (\text{xtract} \ (\forall \beta)) ,
\]

provided $\alpha \notin \mathcal{D}(\forall)$. The above says that we should be able to extract the same terms from the updated graph, except that the variable $\phi_\alpha$ has been replaced with $s \ (\text{xtract} \ (\forall \alpha))^{\beta}$. A straightforward implementation obeying this rule is $\text{upd} \forall \alpha \nu \triangleq \forall + \{ \alpha \rightarrow \nu \}$.

The second operation, $\text{subst}$, has the signature $\text{subst} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{T} ightarrow \mathcal{G}(\mathcal{E})$, where $\mathcal{G} \triangleq \mathcal{X} \rightarrow \mathcal{A}$ are partial functions from variables to addresses. Function $\text{subst}$ takes as arguments a graph $\forall$, a target address $\alpha$, a term $t$, and a mapping $\psi$ (from the free variables in $t$ to addresses in $\forall$). To stick with the jigsaw-puzzle analogy, the effect of $\text{subst}$ is to cut a picture into a collection of pieces, and then connect this collection of pieces to the existing puzzle; from $t$, a collection of nodes is made such that the variables of $t$ are connected to existing nodes, and the root of $t$ is placed at address $\alpha$. As the name suggests, this operation is used to perform what corresponds to substitution in the world
of terms. Formally, it must hold that

$$\nabla' = \text{subst} \nabla \alpha t \psi$$

implies

$$\forall \beta \in \mathcal{D}(\nabla) \cup \{\alpha\} : \text{xtract} \langle \nabla' \beta \rangle = \theta (\text{xtract} \langle \nabla \beta \rangle)$$

where

$$\theta = \{ \phi_{\alpha} \mapsto \{ x \mapsto \text{xtract} \langle \nabla (\psi x) \rangle \mid x \in \mathcal{V}(t) \} \} t \} ,$$

provided $\alpha \notin \mathcal{D}(\nabla)$. The above says that we only allow extensions to graphs, that is, we require that, as long as we stay inside the domain of the graph as it were, the same terms will be extracted after the operation, except that, instead of address $\alpha$ materialising into a variable $\phi_{\alpha}$, $\alpha$ will now materialise into $t$ with the free variables replaced. Note that the operation might have added more than just $\alpha$ to the domain of the graph. See Fig. 1 for a straightforward implementation.

**Example 9.** Take the graph $\nabla$ from Ex. 6, and let $\nabla_0 = \nabla / (\beta)$, shown below to the left. Two legal results w.r.t. (2) of $\text{subst} \nabla_0 \beta (s (\text{ib} x)) \{ x \mapsto \epsilon \}$ are shown below to the right.

![Graphs](image)

In the rest of this section we will be concerned with properties that are independent of the particular implementation of $\text{upd}$ and $\text{subst}$. We will therefore talk about families of operations and functions indexed by such implementations.

**Definition 10 (Realisation).** A realisation $I$ is an implementation of $\text{upd}$ and $\text{subst}$, denoted $\text{upd}_I$ and $\text{subst}_I$, such that $\text{upd}_I$ satisfies (2) and $\text{subst}_I$ satisfies (2).

### 3.3 Semantics

As promised, we can now present a semantics for our little language. The semantics is a small-step operational semantics (Plotkin style [13]) parametrised by a realisation. First we need to translate the initial ground term into graph representation.

**Definition 11 (Initial configuration).** Given a realisation $I$, the function $\text{init}_I \in \mathcal{T} \rightarrow \mathcal{H}(\mathcal{L})$ is defined as $\text{init}_I t \triangleq (\text{subst}_I \emptyset \alpha_0 t \emptyset) \alpha_0$ where $\alpha_0 = \text{FRESH}$.

That is, a new graph is built (on top of an empty graph) such that it represents the initial ground term. Since $t$ is a ground term, it contains no variables, and thus the variable-to-address mapping is empty.
**Theorem 12.** For all realisations \( I \) and ground terms \( t \), \( \text{xtract} \ (\text{init} \ t) = t \).

**Definition 13 (\( \rightarrow \)).** Given a program \( q \) and a realisation \( I \), the binary relation \( \rightarrow_q \subseteq \mathcal{X}(\mathcal{E}) \times \mathcal{X}(\mathcal{E}) \) is defined as the smallest relation satisfying the inference system

\[
\text{(apply)} \quad \frac{f \cdot x^n \overset{\beta}{\Rightarrow} t}{\left(\nabla + \{ \alpha \mapsto f \cdot x^n \} \right) \alpha} \quad \left(\nabla \right) \rightarrow \left(\nabla' \right)
\]

\[
\text{(select)} \quad \frac{\nabla \alpha_0 \rightarrow \alpha_0' \quad \nabla \alpha_0 = c \cdot \beta^m \quad g \cdot (c \cdot x^m) \cdot y^n = t}{\left(\nabla + \{ \alpha \mapsto g \cdot \alpha_0 \cdot \alpha^n \} \right) \alpha} \quad \left(\nabla \right) \rightarrow \left(\nabla' \right)
\]

\[
\text{(dive)} \quad \frac{\nabla \alpha_0 \rightarrow \alpha_0' \quad \nabla \alpha_0 = c \cdot \beta^m \quad g \cdot (c \cdot x^m) \cdot y^n = t}{\left(\nabla + \{ \alpha \mapsto g \cdot \alpha_0 \cdot \alpha^n \} \right) \alpha} \quad \left(\nabla \right) \rightarrow \left(\nabla' \right)
\]

\[
\text{(trans)} \quad \frac{\nabla \alpha} {\nabla \alpha} \rightarrow \left(\nabla' \right)
\]

\[
\text{(const)} \quad \frac{m \leq n \quad \forall i \leq m : \text{xtract} \left(\nabla \alpha_i \right) \in \mathcal{V} \quad \nabla \alpha_m} {\left(\nabla + \{ \alpha \mapsto c \cdot \alpha^n \} \right) \alpha} \quad \left(\nabla \right) \rightarrow \left(\nabla' \right)
\]

\[
\text{(skip)} \quad \frac{\nabla \alpha} {\nabla \alpha} \rightarrow \left(\nabla' \right)
\]

The subscript \( q \cdot I \) has been omitted to avoid clutter.

The inference rules define three relations: \( \rightarrow \), \( \Rightarrow \), and \( \rightarrow \). The relation \( \rightarrow \) relates a configuration containing a function call to the result of the call. Operationally, you can read the rule **apply** as "replace the function call with the function body, in which the variables have been replaced by the arguments to the function"; the **subst** function takes care of both the substitution and the translation from term to graph representation. The **select** and **dive** rules take care of pattern-matching functions: In the former case, the first argument to the function has an outermost constructor, and therefore the call can be replaced by the body of the matching function (similar to the **apply** rule). In the latter case, the first argument to the function does itself contain something that can be **rewritten** by a single step (i.e., a function call); this one-step rewrite is performed, and the result of this rewrite is written back into the graph by an **upd** call. The mutually recursive relations \( \Rightarrow \) and \( \rightarrow \) "dig into" the graph to locate the next
function call to reduce. The skip rule simply skips over indirections and passes control to the trans or const rules, of which the former simply passes control to the above-mentioned → relation, which means that a function call has been located. If a function call has not been located — that is, as long as there are only constructors to the left of the current address — the const rule digs into the leftmost subgraph that can be rewritten (i.e., contains a function call); the rewrite is performed, and the result is written back into the graph by an upd call. The xtract (\( \nu \alpha_i \)) \( \in \mathcal{V} \) part of the premise for const ensures that no rewrites are possible to the left of \( \alpha_{m} \). The relation → is thus responsible for reducing the graph from left to right.

**Theorem 14.** The relation → is well-defined and deterministic.

**Definition 15 (Evaluation).** Given a realisation \( I \), we define the function \( \text{eval}_{q,I} \in \mathcal{T} \to \mathcal{P}(\mathcal{V}) \) by

\[
\text{eval}_{q,I} t \triangleq \{ v \mid (\text{init}_{I} t)_{\text{xtract } \kappa} \in \mathcal{S}(\rightarrow_{q,I}) \land (\text{xtract } \kappa) = v \in \mathcal{V} \} .
\]

We will now state two important properties about the semantics of our little language. The first is a direct consequence of the relation → being deterministic.

**Corollary 16.** A ground term evaluates to at most one value.

A ground term may fail to evaluate to a value for two reasons: Either the computation consists of an infinite number of steps, or the computation “gets stuck” at some point. The former reason is usually called non-termination and is an inherent unpleasantness in any universal programming language. We can, however, circumvent the latter situation by imposing a standard polymorphic type system on our language to reject program/term pairs that will get stuck in a normal form that is not a value. We will not pursue this matter further in this paper, but simply assume that all programs and terms are type correct.

**Definition 17 (Correct).** Given program \( q \) and ground term \( t \), we say that the pair \( \langle q, t \rangle \) is correct if \( (\text{init}_{I} t)_{\text{xtract } \kappa} \in \mathcal{S}(\rightarrow_{q,I}) \) implies \( (\text{xtract } \kappa) \in \mathcal{V} \), for all realisations \( I \).

The second property — and the main reason for the preceding rigour — is that the evaluation of a term (w.r.t. a particular program) always gives us the same result for any realisation.

**Theorem 18 (Realisation independence).** For any program \( q \) and ground term \( t \), if the pair \( \langle q, t \rangle \) is correct, then \( \text{eval}_{q,A} t = \text{eval}_{q,B} t \) for any two realisations \( A \) and \( B \).

**Example 19.** Consider the program \( q \) in Ex.1. The pair consisting of \( q \) and term \( \text{fib} (s(s(s(0)))) \) is correct and evaluates to \( s(s(s(s(0)))) \). The pair consisting of \( q \) and term \( \text{fib} (sA) \) is not correct, since evaluation gets stuck in a configuration representing the term \( \text{aux} A \).
4 Graph machinery

In this section we will present two implementations of subst and upd. The two implementations differ in how they handle sharing of identical subterms.

4.1 Call-by-need

The most straightforward of these implementations is shown in Fig. 1. The explanation of the implementation is as follows.

The upd function simply adds a node to the graph. The subst function calls the auxiliary function saux to ensure that t is converted into graph representation. If the resulting address α' of this conversion is different from the preferred target α, an indirection is made from α to α'. The auxiliary function saux converts a term t into graph representation such that the variables in t are converted into existing addresses in the graph, thus possibly introducing sharing of subgraphs. More precisely, saux adds new nodes to the graph by recursively decomposing t: If t has s as root and n subterms, n fresh addresses are chosen and fed to recursive calls to saux (thus ensuring that the n subterms have been converted into graph representation), and a new node labelled s is created. If t is a variable x, however, no new node is created; instead the provided mapping ψ tells us which existing address to “substitute” for x. The address representing t can thus be different from the preferred target α.

\[
\begin{align*}
\text{subst} & \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{T} \rightarrow \Psi \rightarrow \mathcal{G}(\mathcal{E}) \\
\text{subst} \triangleright \alpha \triangleright t \triangleright \psi & \triangleq \begin{cases} \text{let } (\triangleright \alpha') &= \text{saux } \triangleright \alpha \triangleright t \triangleright \psi \text{ in if } \alpha = \alpha' \text{ then } \triangleright \alpha' \text{ else } \triangleright + \{ \alpha \mapsto \alpha' \} \\
\text{saux } \triangleright \alpha \triangleright t \triangleright \psi & \triangleq \begin{cases} \text{if } t \in X \text{ then } (\triangleright_0 (\psi t)) \\
\text{else let } s \triangleright^n t; (\triangleright X. \alpha.) = \text{saux } \triangleright X. \text{FRESH } t. \psi) \triangleright^n \\
\text{in } (\triangleright (\triangleright_0 + \{ \alpha \mapsto s + \alpha^n \}) \alpha) \\
\text{upd} & \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{N}(\mathcal{E}) \rightarrow \mathcal{G}(\mathcal{E}) \\
\text{upd} \triangleright \alpha \triangleright \nu & \triangleq \triangleright + \{ \alpha \mapsto \nu \}
\end{cases}
\end{align*}
\]

Fig. 1. DAG representation of terms (call-by-need)\(^1\)

It is not hard to see that the implementation in Fig. 1 provides the basis for the standard notion of call-by-need: when used in conjunction with the inference rules defined in Def. 13, all occurrences of the same variable (in a function body) share the same subgraph. When it is necessary to reduce one of these occurrences, all the other occurrences will share the rewrite performed by the inference rules.

4.2 Collapsed jungle

The implementation presented in Fig. 2 is far more interesting, since it will maintain as much sharing as possible. The upd function will never add a new node if
there already exists a similar node. That is, a new node \( \nu \) will not be added to the graph \( \nabla \) at address \( \alpha \) if there already exists a node at \( \beta \) such that \( \text{xtract} \left( \nabla \beta \right) = \text{xtract} \left( \left( \nabla + \{ \alpha \mapsto \nu \} \right) \alpha \right) \). In case such a \( \beta \) exists, we only add an indirection from \( \alpha \) to \( \beta \) to the graph. Furthermore, after adding something to the graph (be it a node or an indirection), the resulting graph is collapsed such that multiple occurrences of similar nodes (in the above sense) are eliminated, and all nodes will have nodes (not indirections) as descendants. Similarly, the \( \text{subst} \) function will make sure that superfluous nodes are not added to the graph. \( \text{subst} \) will call the auxiliary function \( \text{saux} \) to convert the term into graph representation, and if the resulting address is different from the preferred target, an indirection is created, and the (collapsed) graph is returned. The auxiliary function \( \text{saux} \) works like in the call-by-need case, except that a node is not created if a similar one exists.

<table>
<thead>
<tr>
<th>( \text{subst} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{T} \rightarrow \Psi \rightarrow \mathcal{G}(\mathcal{E}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{subst} \nabla \alpha t \psi \overset{\Delta}{=} \begin{cases} \text{saux} \nabla \alpha t \psi &amp; \text{if } \alpha' = \alpha \text{ then } \nabla' \text{ else collapse } (\nabla' + { \alpha \mapsto \alpha' }) \ \end{cases} )</td>
</tr>
<tr>
<td>( \text{saux} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{T} \rightarrow \Psi \rightarrow \mathcal{G}(\mathcal{E}) \times \mathcal{A} )</td>
</tr>
<tr>
<td>( \text{saux} \nabla_0 \alpha t \psi \overset{\Delta}{=} \begin{cases} \text{let } (\nabla_0 \psi t) \sim \alpha' \text{ in } (\nabla_0 \alpha') &amp; \text{if } t \in \mathcal{X} \text{ then} \ \text{else let } s \in \mathcal{F} \text{ in } (\nabla_0 \psi t) \sim \alpha' \text{ in } (\nabla_0 \alpha') &amp; \text{else } \end{cases} )</td>
</tr>
<tr>
<td>( \text{upd} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{G}(\mathcal{E}) )</td>
</tr>
<tr>
<td>( \text{upd} \nabla \alpha (s \cdot \alpha^n) \overset{\Delta}{=} \begin{cases} (\nabla \alpha) \sim \alpha' \text{ in } (\nabla \beta = s \cdot \alpha^n \wedge (\nabla \beta) \sim \alpha') \text{ then collapse } \nabla \text{ else collapse } (\nabla + { \alpha \mapsto s \cdot \alpha^n }) \ \end{cases} )</td>
</tr>
<tr>
<td>( \text{collapse} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{G}(\mathcal{E}) )</td>
</tr>
<tr>
<td>( \text{collapse} \nabla \overset{\Delta}{=} \begin{cases} \text{let } { \alpha \mapsto s \cdot \alpha^n } \cap \mathcal{P}(|\nabla|) \neq \emptyset \text{ then } \nabla \text{ else let } \nabla' + { \alpha \mapsto s \cdot \alpha^n } = \nabla : { \alpha^n } \cap \mathcal{P}(|\nabla|) \neq \emptyset \text{ in } \text{upd} \nabla' \alpha (s \cdot \alpha^n) \ \end{cases} )</td>
</tr>
</tbody>
</table>

**Fig. 2.** Collapsed-jungle representation of terms.¹

In view of our semantic inference rules, collapsing a graph is highly beneficial: Rewriting a single node at \( \alpha \) in a fully collapsed graph will effectively rewrite all subterms identical to the one that can be extracted from \( \alpha \).

**Theorem 20.** The two implementations shown in Figs. 1 and 2 are realisations.

## 5 Transformation

It seems intuitively right that the standard graph machinery (Fig. 1) induces very little administrative overhead, whereas the collapsed-jungle graph machinery (Fig. 2) can be burdensome. We therefore propose a feasible compromise:

¹ The \( (\cdots \cdot \cdots)^n \) is a shorthand for \( n \) equations.
Optimise programs at compile time using collapsed jungles, but use standard graph machinery at run time. As we will see, it is possible to achieve some of the advantages of collapsed-jungle reduction by a source-to-source program transformation.

The program transformation we present here is a variant of supercompilation (Turcín [19, 18]), more specifically positive supercompilation (Sørensen, Glück and Jones [15]) modified to work on jungles instead of terms. The transformation process is divided into two phases. First, a finite model of the program is constructed w.r.t. a term. Second, a new program is extracted from the model.

5.1 Driving

Glück and Klimov [5] call a model of a program a process graph. The nodes in the graph are labelled by terms (i.e., program state), and each successor of a node represents a one-step unfolding. A branch in the process graph thus represents speculative execution of a particular term. Leaves in the process graph represent terms that are fully evaluated. For a particular program \( q \), each full path in a (possibly cyclic) process graph for \( q \) represents a set of actual executions of \( q \), such that the union of all full paths in the process graph includes all possible executions of \( q \).

To keep the process graph manageable, cycles are represented implicitly by leaves containing repetitions of previously seen terms; the graph thus simply becomes a tree.

To construct the process tree, we need to drive the program, that is, speculatively executing non-ground terms. For this purpose we present two modifications of the semantics of the language. The first simply allows variables in terms.

**Definition 21 (Deterministic unfolding, \( B \)).** Let the set of constructor terms \( B \) be given by the grammar \( b ::= x \mid cb^n \). The relation \( \rightarrow \subseteq T \times T \) is defined as \( \rightarrow \) in Def. 13, except that \( V \) is replaced by \( B \) in the \( \text{const} \) rule. \( \square \)

The new relation \( \rightarrow \) is still deterministic, but it allows each reduction step to ignore what corresponds to uninstantiated parts to the left of a redex. With the relation \( \rightarrow \), we can reduce non-ground terms as long as we do not run into redices of the form \( gx^t^n \). To reduce such redices, we need to speculatively try out all possible forms of values of \( x \), according to the definition of \( g \).

**Definition 22 (Speculative unfolding).** The relation \( \vdash \subseteq T \times T \) is defined as \( \rightarrow \), but with the additional rule

\[
\frac{\langle \nabla \alpha_0 \rangle \rightarrow \alpha'_0 \quad \alpha'_0 \notin \beta(\nabla) \quad \beta. = \text{FRESH}^m}{\langle \nabla' \rangle = \text{upd} (\nabla + \{ \alpha \mapsto g \alpha_0 \alpha^n \}) \alpha'_0 (c+\beta^m) \quad g(c x^m)^g^n \triangleq t} \tag{inst}
\]

The relation \( \vdash \) is non-deterministic. When it encounters a stuck redex \( gx^t^n \), it “produces” a new DAG where \( gx^t^n \) has been instantiated to \( g(c x^m) t^n \) for each pattern \( c x^m \) defined by \( g \). Each of these instantiated DAGs will allow
further reduction to take place, since each appropriate right-hand side of \( g \) now can be unfolded.

It is now easy to see how we can create a process tree for a program \( g \) and an initial term \( t \): First, pick some realisation \( I \) and label the root of the process tree by a DAG created from \( t \). Then, repeatedly add new leaves to the process tree by using the relation \( \implies_{g,I} \) to drive existing leaves.

5.2 Generalisations

Unfortunately, creating a process tree in the above manner hardly ever terminates, that is, the process tree will grow unboundedly. But, as shown by Sørensen & Glück [14], a sufficient condition for ensuring that the construction of process trees terminates, is to impose a well-quasi-order on the labels in the process tree.

**Definition 23 (wqo).** A well-quasi-order on a set \( S \) is a reflexive, transitive binary relation \( \leq \) such that, for any infinite sequence \( s_1 \, s_2 \, \ldots \) of elements from \( S \), there are \( i, j \in \mathbb{N} \) such that \( i < j \) and \( s_i \leq s_j \). 

Hence, if we ensure that, for all nodes \( n \) in the process tree, there never exists an ancestor \( a \) of \( n \) such that \( \text{label}(a) \leq \text{label}(n) \), then all branches in the process tree will be finite. Since the process tree is finitely branching, the process tree will be finite (by König’s Lemma).

Sørensen & Glück [14] used the homeomorphic-embedding relation on terms to detect when termination is endangered; we will use this relation in ensuring termination, so a repetition is in order.

**Definition 24.** Let \( s \in \mathcal{E}, x, y \in \mathcal{X}, \) and \( t, u \in \mathcal{T} \). The homeomorphic-embedding relation \( \preceq \) on \( \mathcal{T} \times \mathcal{T} \) is the smallest relation satisfying the inference rules

\[
\begin{align*}
& x \preceq y & & \frac{(t, \leq u)^n}{s \preceq t^n \leq s \, u^n} & & \frac{t \preceq u_i}{t \preceq s \, u^n} & 1 \leq i \leq n
\end{align*}
\]

Since the homeomorphic-embedding relation is a well-quasi-order, it can be used as an indicator for when to stop the development of the process tree, that is, when to stop driving. The question is then what to do, when we need to stop driving. As described by Turchin [19], we need to generalise one of the offending nodes in the process tree, in effect throwing away information that has been acquired during driving. The solution in [14] is to split up such nodes into several parts that can be explored separately. Generalisations thus give rise to branches in the process tree (as do the speculative unfolding).

5.3 Using dags

Since we employ DAGs instead of terms, our generalisation operation needs to split up DAGs. In particular, we want an operation that divide a DAG into two autonomous parts that can be expressed as terms, in order to “reassemble” the state when the transformed program is extracted from the process graph.
Example 25. Consider again the DAG from Ex. 6, shown to the left below.

\[
\begin{array}{c}
\text{data Pair } x y = (x \, y) \\
h x = g (f x) \\
g (u \, v) = ((\text{add} \, u \, v) \, v) \\
f x = ((\text{aux} \, x) \, (\text{fib} \, x))
\end{array}
\]

Splitting up this DAG into two non-trivial DAGs can be done as indicated in the middle. To the right is shown how such a split can be represented in the term world, interpreting the roots of lower half of the DAG as a tuple of terms. □

Spitting up a DAG thus naturally gives rise to the notion of a root list of a DAG. The example should give enough intuition to support the following definitions.

Definition 26 (Roots, ports, subdags, and proper splits).

1. Every DAG \( \mathcal{V} \) is implicitly accompanied a finite root list, \( \text{roots} (\mathcal{V}) \in \mathcal{A}^* \).
2. The ports of a DAG \( \mathcal{V} \) is the set of addresses outside the domain of \( \mathcal{V} \) that are reachable through the roots of \( \mathcal{V} \):
   \[
   \text{ports}(\mathcal{V}) \triangleq \{ \beta \in \mathcal{A} \mid \beta \notin \mathcal{D}(\mathcal{V}) \land \exists \alpha \in \text{roots}(\mathcal{V}) : \alpha \rightarrow^* \beta \}.
   \]
3. A DAG \( \mathcal{V}' \) is a subdag of a DAG \( \mathcal{V} \), denoted \( \mathcal{V}' \leq \mathcal{V} \), if \( \mathcal{V}' \subseteq \mathcal{V} \) and
   \[
   \forall \alpha \in \mathcal{D}(\mathcal{V}) \setminus \mathcal{D}(\mathcal{V}') : (\alpha \rightarrow \beta \text{ implies } \beta \notin \{ \beta \mid \alpha \in \text{roots}(\mathcal{V}') \rightarrow^\dagger \beta \})
   \]
4. The pair \( (\mathcal{V}', \mathcal{V}'') \) is a proper split of \( \mathcal{V} \) if \( \mathcal{V} = \mathcal{V}' + \mathcal{V}'' \), \( \mathcal{V}'' \leq \mathcal{V} \), and \( \mathcal{V}' \neq \emptyset \neq \mathcal{V}'' \). □

Informally, all nodes external to a subdag can only reach nodes in the subdag through the roots of the subdag. That is, if \( \mathcal{V}' \leq \mathcal{V} \), then \( \mathcal{V}' \) can be “carved” out of \( \mathcal{V} \), such that \( \mathcal{V} = \mathcal{V}' + \mathcal{V}'' \) and \( \mathcal{V}'' \) interacts with \( \mathcal{V}' \) only through the roots of \( \mathcal{V}' \). A proper split is then a division of a DAG into two non-trivial parts. The split in Ex.25 is proper.

We will, however, use such a split operation as a last resort. A more sophisticated generalisation can be achieved when the offending node \( n \) in the process tree has an ancestor \( a \) such that \( n \) is reducible to \( a \). Informally, \( \mathcal{V} \) is reducible to \( \mathcal{V}_1 \), if the terms in \( \mathcal{V} \) can be reconstructed by carving out a subdag \( \mathcal{V}_3 \) (of \( \mathcal{V} \)) and connecting it with \( \mathcal{V}_1 \) via a set of indirections \( \mathcal{V}_2 \).

Definition 27 (terms, reducible).

1. \( \text{terms}(\mathcal{V}) \triangleq [\text{extract} (\mathcal{V} \, \alpha)] \mid \alpha \leftarrow \text{roots}(\mathcal{V})] \).
2. \( \mathcal{V} \) is reducible to \( \mathcal{V}_1 \) by \( \mathcal{V}_2 = \{ (\alpha \mapsto \beta)^n \} \) and \( \mathcal{V}_3 \), if
   (a) \( \text{ports}(\mathcal{V}_1) = \{ \alpha^n \} \),
(b) \( \text{roots}(\nabla_2) = [\alpha^n] \),
(c) \( \text{roots}(\nabla_3) = [\beta | \beta \leftarrow \text{ports}(\nabla_2)] \),
(d) \( \nabla_3 \leq \nabla \), and
(e) \( \text{terms}(\nabla) = \text{terms}(\nabla_1 + \nabla_2 + \nabla_3) \). \( \Box \)

**Example 28.** Assume that the DAG \( \nabla_1 \) in the middle is an ancestor of the DAG \( \nabla \) to the left.

The state of \( \nabla \) can be generalised into a function \( b \) by calling the function \( a \), representing state \( \nabla_1 \), by providing the arguments constructed by functions \( c \) and \( d \), representing the indirections \( \nabla_2 \) and subdag \( \nabla_3 \) to the right. \( \Box \)

We have now established some means of generalising DAGs (accompanied with root lists), and alluded to how code can be generated from such generalisations. To ensure termination of transformation, it is crucial that every generalisation breaks down a DAG into strictly smaller components.

**Remark 29.** For code-generation purposes, it is furthermore beneficial to strip every DAG of its outermost constructors and indirections by yet another generalisation step. For this presentation, however, such operation is not needed, and we therefore leave out the details.

The point of reducing a DAG to an ancestor is almost obvious: Only the subdag that has been carved out needs to be driven further. This property calls for a definition of process trees and finished nodes.

**Definition 30 (Process trees, labels, leaves, ancestors, finished).** A **process tree** \( \tau \) is a non-empty tree labelled with DAGs. For a particular node \( \nu \) in \( \tau \), the **label of** \( \nu \) is denoted \( \text{label}(\nu) \), and the set of ancestors is the set of proper predecessors of \( \nu \) in \( \tau \), denoted \( \text{anc}(\tau, \nu) \). The leaves of \( \tau \) are denoted by \( \text{leaves}(\tau) \). A node \( \nu \) in \( \tau \) is **finished** if one of the following holds.

1. \( \nu \notin \text{leaves}(\tau) \).
2. \( \exists \mu \in \text{anc}(\tau, \nu) : \text{label}(\nu) = \text{label}(\mu) \).
3. \( \text{terms}(\text{label}(\nu)) \in B^* \).

A process tree is **finished**, if all nodes are finished. \( \Box \)
That is, a node \( \nu \) in a process tree is finished if \( \nu \) is an interior node, if \( \nu \) is a repetition, or if there are no function symbols left to drive in \( \nu \).

It remains to define exactly when generalisations are needed. The following quasi-order seems to be desirable.

**Definition 31.** We say that \( \nabla \) is *embedded* in \( \nabla' \), denoted \( \nabla \preceq \nabla' \), if both

\[
\forall t \in \text{terms}(\nabla) : \exists u \in \text{terms}(\nabla') : t \leq u \\
\forall u \in \text{terms}(\nabla') : \exists t \in \text{terms}(\nabla) : t \leq u .
\]

We define embeddings \( \nabla \nabla' \triangleq \{ \alpha \in \mathcal{P}(\nabla') | \exists \beta \in \text{roots}(\nabla) : (\text{xtract} \{ \nabla \beta \}) \preceq (\text{xtract} \{ \nabla' \alpha \}) \} . \)

**Conjecture 32.** \( \preceq \) is a well-quasi-order.

**Remark 33.** As of this writing, we have not been able establish a proof of the above conjecture. If the conjecture is false, another suitable well-quasi-order needs to be invented. Leuschel [10] describes why well-quasi-orders are preferable over well-founded orders.

```
input: program q, term t, and a realisation I
output: the process tree \( \tau \)
let \( \alpha_0 = \text{FRESH} \)
let tree \( \tau \) consist of a single node labelled \( (\text{sub}t, \emptyset, \alpha_0, t, \phi^{-1}) \) and roots \( \{\alpha_0\} \)
while \( \tau \) is unfinished do
  let \( \nu \) be an unfinished node with \( \nabla = \text{label}(\nu) \)
  if \( \not\exists \mu \in \text{relanc}(\tau, \nu) : \text{label}(\mu) \preceq \nabla \)
    then let \( \alpha \in \text{roots}(\nabla) : \exists \nabla' : (\nabla \alpha) \equiv_q^{q^t} \nabla' \)
      add children to \( \nu \) with labels \( [\nabla'] \)
    else let \( \mu \in \text{relanc}(\tau, \nu) : \nabla_1 = \text{label}(\mu) \wedge \nabla_1 \preceq \nabla \)
      if \( \nabla \) is reducible to \( \nabla_1 \) by \( \nabla_2 \) and \( \nabla_3 \)
        then add three children to \( \nu \) with labels \( \nabla_1, \nabla_2, \) and \( \nabla_3 \)
      else if \( \exists (\nabla_2 \nabla_3) : (\nabla_2 \nabla_3) \) is a proper split of \( \nabla \) and
        \( \text{ports}(\nabla_2) \cap (\text{embeddings} \nabla_1 \backslash \nabla) \neq \emptyset \)
        then add two children to \( \nu \) with labels \( \nabla_2 \) and \( \nabla_3 \)
      else let \( (\nabla_2 \nabla_3) \) be a proper split of \( \nabla_1 \)
        replace all subtrees of \( \mu \) with two children labelled \( \nabla_2 \) and \( \nabla_3 \)
```

**Fig. 3.** The process-trees construction algorithm.

An algorithm for developing process trees is depicted in Fig. 3, assuming for the moment that \( \text{relanc}(\tau, \nu) = \text{anc}(\tau, \nu) \).
It turns out, however, that scrutinising all ancestors is too conservative: too many generalisations happen. Firstly, when a DAG $\nabla$ is speculatively unfolded to a DAG $\nabla'$ by an instantiation step, it is always the case that $\nabla \preceq \nabla'$. Secondly, an instantiation step will give rise to a series of deterministic unfoldings (Turčin [19] calls these *transient reductions*). It is well known from partial evaluation [9] and deforestation [21] that such deterministic unfoldings are very beneficial, in that they are invariants in the program $q$.

We will therefore adapt a notion of *relevant* ancestors, as introduced in Sørensen & Glück [16].

**Definition 34 (relevant ancestors).** Let $\nu$ by a node in a process tree $\tau$.

1. $\nu$ is *generalised*, if its children have been added by a generalisation step.
2. $\nu$ is *global*, if its parent node is generalised, and/or if $\nu$'s children can be produced by an instantiation step (i.e., $\nu$ cannot be unfolded by $\mapsto$ alone).
3. $\nu$ is *local*, if it is neither generalised nor global (i.e., $\nu$ can be unfolded by $\mapsto$).
4. The set of *immediate local ancestors* of $\nu$, $\text{locanc}(\tau, \nu)$, is the set of local nodes in the longest branch of local nodes $\mu_1 \ldots \mu_n$ in $\tau$ such that $\mu_n$ is the parent of $\nu$.
5. The set of *relevant ancestors* of $\nu$ in $\tau$ is defined as

\[
\text{relanc}(\tau, \nu) \triangleq \begin{cases} 
\{ \mu | \mu \in \text{anc}(\tau, \nu) \land \mu \text{ is global} \} & \text{if } \nu \text{ is global} \\
\text{locanc}(\tau, \nu) & \text{if } \nu \text{ is local}
\end{cases}
\]

**Conjecture 35.** The algorithm in Fig. 3 terminates for all programs.

Informally, the restriction to relevant ancestors is safe by the following reasoning. There cannot be a branch with an infinite number of consecutive local nodes, since then there would be an embedding, resulting in a generalisation, thus creating a global node. Since every node only can be generalised once, breaking it into strictly smaller pieces, the process tree stabilises (as a Cauchy sequence). The proposed algorithm is thus an instance of what Sørensen calls an *abstract program transformer* [17]. However, the above remains a conjecture, in the light of the missing proof of Conj.32.

**Theorem 36.** The algorithm in Fig. 3 results in a program that is equivalent to the original program.

The efficiency of the transformed program depends on the particular realisation $I$ used by the unfolding rules.\footnote{To some extent, the efficiency also depends on the treatment of sharing between the outermost constructors; the produced code must carefully mimic such sharing.} We have not been able to establish proofs of the efficiency of the transformed program with respect to $I$, but it seems likely that both of the realisations in Figs.1 and 2 will guarantee that the transformed program is at least as efficient as the original program.
**Example 37.** Let \( q \) be the Fibonacci Number program in Fig. 1, let \( t = \texttt{fib} x \). If collapsed jungles are chosen as the underlying reduction strategy, the algorithm in Fig. 3 will produce the process tree depicted in Fig. 4. A program very similar to the following can be extracted from the process tree:

\[
\begin{align*}
\textbf{data Nat} &\quad = 0 \mid s \text{Nat} \\
\textbf{data Pair} x y &\quad = \langle x, y \rangle \\
a \mathbin{0} &\quad = s \mathbin{0} \\
a \langle s x \rangle &\quad = b x \\
b \mathbin{0} &\quad = s \mathbin{0} \\
b \langle s y \rangle &\quad = e \langle d y \rangle \\
c \langle x y \rangle &\quad = e \mathbin{x} y \\
d \mathbin{0} &\quad = \langle \langle s \mathbin{0} \rangle (s \mathbin{0}) \rangle \\
d \langle s x \rangle &\quad = f \langle d x \rangle \\
e \mathbin{0} y &\quad = y \\
e \langle s x \rangle y &\quad = s \langle e \mathbin{x} y \rangle \\
f \langle x y \rangle &\quad = g \mathbin{x} y \\
g \mathbin{0} y &\quad = \langle y \mathbin{0} \rangle \\
g \langle s x \rangle y &\quad = h \langle i \langle g \mathbin{x} y \rangle \rangle \\
h \mathbin{x y} &\quad = \langle \langle s y \rangle x \rangle \\
i \mathbin{x y} &\quad = \langle \langle s y \rangle x \rangle
\end{align*}
\]

The tuples in this program stem from multiple roots. Observe that, in comparison to the original, the transformed program avoids making an exponential number of calls. \( \Box \)

## 6 Conclusion and Related Work

The benefits of deforestation and supercompilation are well illustrated in the literature, and advances in ensuring termination of these (and similar) transformations have greatly improved their potential as automatic, off-the-shelf optimisation techniques. One problem, however, remains in making these techniques suitable for inclusion in the standard tool-box employed by compiler writers: It is in general not possible to ensure that a transformed program is at least as efficient as the original program, without imposing severe (usually syntactic) restrictions on the original programs.

In this paper we have tried to formulate a version of positive supercompilation that addresses the concern of ensuring efficiency of the transformed program. The key ingredient in this formulation is the return to viewing terms as graphs.

In the first part of this paper, we have shown that, for a small function-oriented programming language, any graph-reduction implementation obeying two reasonable rules will lead to the same semantics. We have given two examples of such implementations, one similar to call-by-need, and one similar to collapsed-jungle evaluation.

Wadsworth [22] invented call-by-need for the pure \( \lambda \)-calculus, and proved that normal-order (call-by-need) graph reduction is at least as efficient as normal-order term reduction for a certain subset of graphs representing \( \lambda \)-terms, and he devised an algorithm for performing normal-order reduction. Hoffmann & Plump [7], the main source of inspiration for this research, have proved that term rewrite systems could be translated into hypergraph replacement systems. They define the notion of fully-collapsed jungles in terms of morphisms on graphs, and they show uniqueness of such fully-collapsed graphs. The collapse-function given in our second realisation of graph reduction is basically an implementation of
their fold-morphism. Their main focus, however, is on showing that confluence and termination is preserved for a large class of term rewrite systems.

In the second part of this paper, we have presented a version of supercompilation that — when using collapsed jungles as the underlying representation — can give some of the speedups that collapsed-jungle evaluation can give, but without any run-time overhead. In this respect, we have effectively achieved to perform tupling (Pettorossi [12] and Chin [2]), an aggressive, semi-automatic program transformation based on the unfold/fold framework (Burstall & Darlington [1]). The key ingredient in tupling is to discover a set of progressive cuts [12] in the call graph for a program, and automatic search procedures for such cuts have been investigated intensively. In particular, Pettorossi, Pietropoli & Proietti [11] manipulate DAGs in a fashion that is very similar to our notion of a proper split.\(^3\) It seems that we are able to synchronise common calls, because we use a local/global unfolding strategy similar to what is used in partial deduction (see e.g., Gallagher [4] or De Schreye, et al. [3]).

Further work needs to be done in three directions. Firstly, we need to prove the efficiency and correctness properties conjectured in this paper. Secondly, we want to investigate the exact relationship between tupling and graph-based supercompilation. Thirdly, to establish empirical results, an implementation of the presented transformer is under construction. In the future, we hope to bootstrap the transformer, in the sense of expressing it in terms of the subject language. Having done this, it will be possible to experiment with self-application (e.g., as described by Jones, Sestoft, and Sondergaard [8] or Turchin [20]).

Acknowledgements

I want to thank Robert Glück, Neil D. Jones, Michael Leuschel, Oege de Moor, Henning Niss, and (certainly not least) Morten Heine Sørensen for enlightening discussions. I also want to thank the anonymous referees for their detailed comments and suggestions for improvements.

References


\(^3\) Maurizio Proietti kindly provided this paper. Unfortunately it arrived very close to the deadline, so the precise relationship is unclear as of this writing.
Fig. 4. Process tree of the Fibonacci program. Global nodes are shaded.


