Driving in the Jungle*

Jens Peter Secher

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

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 fulfils 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 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 $\langle a\,b\,c\,d\rangle$ 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 $\mathscr{P}(S)$; 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 $(x.)^5$ for such a sequence: The superscript "5" 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 all together, and, for example, write x^n instead of $(x.)^n$. When this kind of notation is used in several layers, the innermost part is expanded first. Hence $\{(x.\mapsto t^2)^n\}$ means $\{(x_1\mapsto t_1\,t_2)\cdots(x_{n-1}\mapsto t_1\,t_2)\,(x_n\mapsto t_1\,t_2)\}$; the empty sequence is also allowed, so $\{(x.\mapsto t.)^0\}$ means $\{\}=\varnothing$.

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

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:

$$\begin{array}{l|l}
\textbf{data}\,\mathrm{Nat} = 0 \mid \mathrm{S}\,\mathrm{Nat} & \\
\mathit{fib}\,0 = \mathrm{S}\,0 & \mathit{aux}\,(\mathrm{S}\,y) = \mathit{add}\,(\mathit{aux}\,y)\,(\mathit{fib}\,y) \\
\mathit{fib}\,(\mathrm{S}\,x) = \mathit{aux}\,x & \mathit{add}\,0\,y = y \\
\mathit{aux}\,0 = \mathrm{S}\,0 & \mathit{add}\,(\mathrm{S}\,x)\,y = \mathrm{S}\,(\mathit{add}\,x\,y)
\end{array}$$ □

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

$$\begin{array}{lll}
\text{(program)} & \mathcal{Q} \ni q ::= d^m & \text{(definitions)} \\
\text{(definition)} & \mathcal{D} \ni d ::= f\,x^n = t \mid (g\,p.\,x^n = t.)^m & \text{(function/matcher)} \\
\text{(pattern)} & \mathcal{P} \ni p ::= c\,x^n & \text{(flat pattern)} \\
\text{(term)} & \mathcal{T} \ni t ::= x \mid c\,t^n \mid f\,t^n \mid g\,t^m & \\
 & & \text{(variable/construction/application/match)} \\
\text{(value)} & \mathcal{V} \ni v ::= c\,v^n &
\end{array}$$

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

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 $\psi = \{(x. \mapsto t.)^n\} \in \mathcal{X} \to \mathcal{T}$ from variables to terms, we denote by $\theta = \{\!\{(x. \mapsto t.)^n\}\!\} \in \mathcal{T} \to \mathcal{T}$ the *substitution induced by* ψ. □

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 α and β 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 ν. We will use the shorter notation $\nu = s \dashv \alpha^n$ instead of writing $\nu = \langle s\ [\alpha^n] \rangle$.
2. We denote by *directed graphs over* S the set $\mathcal{A} \to (\mathcal{N}(S) + \mathcal{A})$ of partial mappings from addresses to nodes/addresses. Any directed graph G induces a binary relation $\multimap \subseteq \mathcal{A} \times \mathcal{A}$ defined by $\beta \multimap \alpha_i$ iff $G\,\beta = \alpha_i$ or $G\,\beta = s \dashv \alpha^n \wedge i \leq n$.
3. We denote by *acyclic directed graphs over* S the set $\mathcal{G}(S) \subseteq \mathcal{A} \to (\mathcal{N}(S) + \mathcal{A})$ of acyclic graphs (i.e., G acyclic iff $\nexists \alpha \in \mathcal{D}(G) : \alpha \overset{+}{\multimap} \alpha$). We let ∇ range over the set of acyclic graphs.
4. We denote by *configurations over* S the set $\mathcal{K}(S) \triangleq \mathcal{G}(S) \times \mathcal{A}$, ranged over by κ.
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

$$\|\nabla\| \triangleq \{\alpha \mapsto \beta \mid \nabla\,\alpha = \beta \in \mathcal{A}\}\ .$$

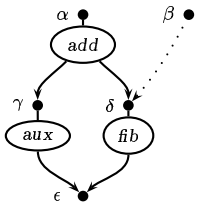
   The relation $\rightsquigarrow \subseteq \mathcal{K}(\mathcal{E}) \times \mathcal{A}$ is defined inductively by

$$\begin{array}{l} \langle \nabla\,\alpha \rangle \rightsquigarrow \alpha\ \text{, if } \alpha \notin \mathcal{D}(\|\nabla\|) \\ \langle \nabla\,\alpha \rangle \rightsquigarrow \alpha'\ \text{, if } \|\nabla\|\,\alpha = \alpha'' \text{ and } \langle \nabla\,\alpha'' \rangle \rightsquigarrow \alpha'\ . \end{array}$$

6. Let A be a set of addresses. We let $\nabla/_A$ denote the graph ∇ restricted to $\mathcal{D}(\nabla) \setminus A$.
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 $\{0\ \mathrm{S}\ add\ fib\ aux\}$ be a set of symbols and $\{\alpha\,\gamma\,\delta\,\beta\,\epsilon\}$ a set of addresses, and consider the graph

$$\nabla = \left\{ \begin{array}{l} \alpha \mapsto add \dashv \gamma\,\delta \\ \gamma \mapsto aux \dashv \epsilon \\ \delta \mapsto fib \dashv \epsilon \\ \beta \mapsto \delta \end{array} \right\}\ .$$



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

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 ϕ_α we denote the variable $\phi\,\alpha$. The function $xtract \in \mathscr{K}(\mathcal{E}) \to \mathcal{T}$ is defined inductively by

$$\begin{array}{ll} xtract\ \langle \nabla\ \alpha \rangle & \triangleq \phi_\alpha, \text{ if } \alpha \notin \mathscr{D}(\nabla) \\ xtract\ \langle (\nabla + \{\alpha \mapsto \beta\})\ \alpha \rangle & \triangleq xtract\ \langle \nabla\ \beta \rangle \\ xtract\ \langle (\nabla + \{\alpha \mapsto s \dashv \beta^n\})\ \alpha \rangle & \triangleq s\ (xtract\ \langle \nabla\ \beta. \rangle)^n \end{array} \qquad \square$$

Example 8 (Extraction). Assume $\phi_\epsilon = x$, and consider the graph ∇ from Ex. 6; we have that $xtract\ \langle \nabla\ \beta \rangle = fib\ x$ and $xtract\ \langle \nabla\ \alpha \rangle = add\ (aux\ x)\ (fib\ x)$. $\square$

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, *upd*, has the signature $upd \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathscr{N}(\mathcal{E}) \to \mathscr{G}(\mathcal{E})$. It takes as arguments a graph ∇, a target address α, and a node ν; Intuitively, calling *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, *upd* updates ∇ with ν at α such that it connects with existing nodes in ∇. Formally, it must hold that

$$\begin{array}{c} \nabla' = upd\ \nabla\ \alpha\ (s \dashv \alpha^n) \\ \text{implies} \\ \forall \beta \in \mathcal{A} : xtract\ \langle \nabla'\ \beta \rangle = (\{\!\{ \phi_\alpha \mapsto s\ (xtract\ \langle \nabla\ \alpha. \rangle)^n \}\!\})\ (xtract\ \langle \nabla\ \beta \rangle) \ , \end{array} \qquad (\natural)$$

provided $\alpha \notin \mathscr{D}(\nabla)$. The above says that we should be able to extract the same terms from the updated graph, except that the variable ϕ_α has been replaced with $s\ (xtract\ \langle \nabla\ \alpha. \rangle)^n$. A straightforward implementation obeying this rule is $upd\ \nabla\ \alpha\ \nu \triangleq \nabla + \{\alpha \mapsto \nu\}$.

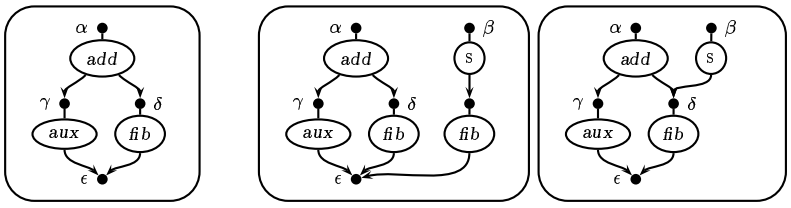
The second operation, *subst*, has the signature $subst \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathscr{G}(\mathcal{E})$, where $\Psi \triangleq \mathcal{X} \to \mathcal{A}$ are partial functions from variables to addresses. Function *subst* takes as arguments a graph ∇, a target address α, a term t, and a mapping ψ (from the free variables in t to addresses in ∇). To stick with the jigsaw-puzzle analogy, the effect of *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 α. As the name suggests, this operation is used to perform what corresponds to substitution in the world

of terms. Formally, it must hold that

$$\nabla' = \mathit{subst}\ \nabla\ \alpha\ t\ \psi$$
$$\text{implies}$$
$$\forall \beta \in \mathscr{D}(\nabla) \cup \{\alpha\} : \mathit{xtract}\ \langle \nabla'\ \beta \rangle = \theta\ (\mathit{xtract}\ \langle \nabla\ \beta \rangle) \qquad (\sharp)$$
$$\text{where}$$
$$\theta = \{\!\{ \phi_\alpha \mapsto \{\!\{ x \mapsto \mathit{xtract}\ \langle \nabla\ (\psi\ x) \rangle \mid x \in \mathscr{V}(t) \}\!\}\ t \}\!\}\ ,$$

provided $\alpha \notin \mathscr{D}(\nabla)$. 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 α materialising into a variable ϕ_α, α will now materialise into t with the free variables replaced. Note that the operation might have added more than just α to the domain of the graph. See Fig. 1 for a straightforward implementation.

Example 9. Take the graph ∇ from Ex. 6, and let $\nabla_0 = \nabla/_{\{\beta\}}$, shown below to the left. Two legal results w.r.t. ($\sharp$) of $\mathit{subst}\ \nabla_0\ \beta\ (\mathrm{S}\ (\mathit{fib}\ x))\ \{x \mapsto \epsilon\}$ are shown below to the right.



□

In the rest of this section we will be concerned with properties that are independent of the particular implementation of *upd* and *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 *upd* and *subst*, denoted upd_I and subst_I, such that upd_I satisfies ($\natural$) and subst_I satisfies ($\sharp$). □

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 $\mathit{init}_I \in \mathcal{T} \to \mathscr{K}(\mathcal{E})$ is defined as $\mathit{init}_I\ t \triangleq \langle (\mathit{subst}_I \varnothing\ \alpha_0\ t \varnothing)\ \alpha_0 \rangle$ where $\alpha_0 =$ 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, $xtract\,(init_I\ t) = t$.*

Definition 13 ($\underset{q\cdot I}{\longrightarrow}$). Given a program q and a realisation I, the binary relation $\underset{q\cdot I}{\longrightarrow} \subseteq \mathscr{K}(\mathcal{E}) \times \mathscr{K}(\mathcal{E})$ is defined as the smallest relation satisfying the inference system

$$\text{(apply)}\ \frac{f\,x^n \stackrel{q}{=} t \qquad \nabla' = subst\ \nabla\ \alpha\ t\ \{(x. \mapsto \alpha.)^n\}}{\langle(\nabla + \{\alpha \mapsto f \dashv \alpha^n\})\ \alpha\rangle \hookrightarrow \langle\nabla'\ \alpha\rangle} \qquad \boxed{\langle\nabla\ \alpha\rangle \hookrightarrow \langle\nabla'\ \alpha\rangle}$$

$$\text{(select)}\ \frac{\langle\nabla\ \alpha_0\rangle \rightsquigarrow \alpha_0' \qquad \nabla\ \alpha_0' = c \dashv \beta^m \qquad g\ (c\,x^m)\ y^n \stackrel{q}{=} t \qquad \nabla' = subst\ \nabla\ \alpha\ t\ \{(x. \mapsto \beta.)^m\ (y. \mapsto \alpha.)^n\}}{\langle(\nabla + \{\alpha \mapsto g \dashv \alpha_0\ \alpha^n\})\ \alpha\rangle \hookrightarrow \langle\nabla'\ \alpha\rangle}$$

$$\text{(dive)}\ \frac{\langle\nabla\ \alpha_0\rangle \rightsquigarrow \alpha_0' \qquad \langle\nabla\ \alpha_0'\rangle \hookrightarrow \langle\nabla'\ \alpha_0'\rangle}{\langle(\nabla + \{\alpha \mapsto g \dashv \alpha_0\ \alpha^n\})\ \alpha\rangle \hookrightarrow \langle(upd\ \nabla'\ \alpha\ (g \dashv \alpha_0\ \alpha^n))\ \alpha\rangle}$$

$$\text{(trans)}\ \frac{\langle\nabla\ \alpha\rangle \hookrightarrow \langle\nabla'\ \alpha\rangle}{\langle\nabla\ \alpha\rangle \rightarrowtail \langle\nabla'\ \alpha\rangle} \qquad \boxed{\langle\nabla\ \alpha\rangle \rightarrowtail \langle\nabla'\ \alpha\rangle}$$

$$\text{(const)}\ \frac{m \le n \qquad \forall i < m : xtract\ \langle\nabla\ \alpha_i\rangle \in \mathcal{V} \qquad \langle\nabla\ \alpha_m\rangle \longrightarrow \langle\nabla'\ \alpha_m\rangle}{\langle(\nabla + \{\alpha \mapsto c \dashv \alpha^n\})\ \alpha\rangle \rightarrowtail \langle(upd\ \nabla'\ \alpha\ (c \dashv \alpha^n))\ \alpha\rangle}$$

$$\text{(skip)}\ \frac{\langle\nabla\ \alpha\rangle \rightsquigarrow \alpha' \qquad \langle\nabla\ \alpha'\rangle \rightarrowtail \langle\nabla'\ \alpha'\rangle}{\langle\nabla\ \alpha\rangle \longrightarrow \langle\nabla'\ \alpha\rangle} \qquad \boxed{\langle\nabla\ \alpha\rangle \longrightarrow \langle\nabla'\ \alpha\rangle}$$

The subscript $q \cdot I$ has been omitted to avoid clutter. □

The inference rules define three relations: $\hookrightarrow$, $\rightarrowtail$, and $\longrightarrow$. The relation $\hookrightarrow$ 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 $\rightarrowtail$ and $\longrightarrow$ "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 $\hookrightarrow$ 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* $\langle \nabla\ \alpha_i \rangle \in \mathcal{V}$ part of the premise for *const* ensures that no rewrites are possible to the left of α_m. The relation $\longrightarrow$ is thus responsible for reducing the graph from left to right.

Theorem 14. *The relation $\longrightarrow$ is well-defined and deterministic.*

Definition 15 (Evaluation). Given a realisation I, we define the function $eval_{q\cdot I} \in \mathcal{T} \to \mathscr{P}(\mathcal{V})$ by

$$eval_{q\cdot I}\, t \triangleq \{v \mid (init_I\, t) \xrightarrow[q\cdot I]{*} \kappa \in \mathscr{E}(\xrightarrow[q\cdot I]{}) \wedge (xtract\, \kappa) = v \in \mathcal{V}\}\ . \qquad \square$$

We will now state two important properties about the semantics of our little language. The first is a direct consequence of the relation $\longrightarrow$ 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 $(init_I\, t) \xrightarrow[q\cdot I]{*} \kappa \in \mathscr{E}(\xrightarrow[q\cdot I]{})$ implies $(xtract\, \kappa) \in \mathcal{V}$, for all realisations I. $\square$

The second property — and the main reason for the preceding rigor — 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 $eval_{q\cdot A}\, t = eval_{q\cdot B}\, t$ for any two realisations A and B.*

Example 19. Consider the program q in Ex. 1. The pair consisting of q and term *fib* (S (S (S (S 0)))) is correct and evaluates to S (S (S (S (S 0)))). The pair consisting of q and term *fib* (S A) is not correct, since evaluation gets stuck in a configuration representing the term *aux* A. $\square$

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.

$$
\begin{array}{l}
\mathit{subst} \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathscr{G}(\mathcal{E}) \\
\mathit{subst}\ \nabla\ \alpha\ t\ \psi \triangleq \textbf{let}\ \langle \nabla'\ \alpha' \rangle = \mathit{saux}\ \nabla\ \alpha\ t\ \psi\ \textbf{in if}\ \alpha = \alpha'\ \textbf{then}\ \nabla'\ \textbf{else}\ \nabla + \{\alpha \mapsto \alpha'\} \\
\mathit{saux} \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathscr{K}(\mathcal{E}) \\
\mathit{saux}\ \nabla_0\ \alpha\ t\ \psi \triangleq \textbf{if}\ t \in \mathcal{X}\ \textbf{then}\ \langle \nabla_0\ (\psi\ t) \rangle \\
\qquad\qquad \textbf{else let}\ s\ t^n = t;\ (\langle \nabla_\cdot\ \alpha_\cdot \rangle = \mathit{saux}\ \nabla_{\cdot - 1}\ \text{FRESH}\ t_\cdot\ \psi)^n \\
\qquad\qquad\quad \textbf{in}\ \langle (\nabla_n + \{\alpha \mapsto s \dashv \alpha^n\})\ \alpha \rangle \\
\mathit{upd} \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathscr{N}(\mathcal{E}) \to \mathscr{G}(\mathcal{E}) \\
\mathit{upd}\ \nabla\ \alpha\ \nu \quad \triangleq \nabla + \{\alpha \mapsto \nu\}
\end{array}
$$

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 ν will not be added to the graph ∇ at address α if there already exists a node at β such that $xtract\ \langle \nabla\ \beta \rangle = xtract\ \langle (\nabla + \{\alpha \mapsto \nu\})\ \alpha \rangle$. In case such a β exists, we only add an indirection from α to β 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 *subst* function will make sure that superfluous nodes are not added to the graph. *subst* will call the auxiliary function *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 *saux* works like in the call-by-need case, except that a node is not created if a similar one exists.

$$
\begin{array}{l}
subst \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathscr{G}(\mathcal{E}) \\
subst\ \nabla\ \alpha\ t\ \psi \quad \triangleq \mathbf{let}\ \langle \nabla'\ \alpha' \rangle = saux\ \nabla\ \alpha\ t\ \psi \\
\qquad \mathbf{in\ if}\ \alpha' = \alpha\ \mathbf{then}\ \nabla'\ \mathbf{else}\ collapse\ (\nabla' + \{\alpha \mapsto \alpha'\}) \\
saux \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathscr{G}(\mathcal{E}) \times \mathcal{A} \\
saux\ \nabla_0\ \alpha\ t\ \psi \quad \triangleq \mathbf{if}\ t \in \mathcal{X}\ \mathbf{then\ let}\ \langle \nabla_0\ (\psi\ t) \rangle \rightsquigarrow \alpha'\ \mathbf{in}\ \langle \nabla_0\ \alpha' \rangle \\
\qquad \mathbf{else\ let}\ s\ t^n = t;\ (\langle \nabla.\ \alpha. \rangle = saux\ \nabla_{.-1}\ \textsc{fresh}\ t.\ \psi)^n \\
\qquad\qquad \mathbf{in\ if}\ \exists \beta \in \mathscr{D}(\nabla_n) : \nabla_n\ \beta = s \dashv \beta^n \wedge (\langle \nabla_n\ \beta. \rangle \rightsquigarrow \alpha.)^n \\
\qquad\qquad\quad \mathbf{then}\ \langle \nabla_n\ \beta \rangle\ \mathbf{else}\ \langle (\nabla_n + \{\alpha \mapsto s \dashv \alpha^n\})\ \alpha \rangle \\
upd \in \mathscr{G}(\mathcal{E}) \to \mathcal{A} \to \mathscr{N}(\mathcal{E}) \to \mathscr{G}(\mathcal{E}) \\
upd\ \nabla\ \alpha\ (s \dashv \alpha^n) \triangleq \mathbf{let}\ (\langle \nabla\ \alpha. \rangle \rightsquigarrow \alpha.')^n\ \mathbf{in\ if}\ \exists \beta : \nabla\ \beta = s \dashv \beta.^n \wedge (\langle \nabla\ \beta. \rangle \rightsquigarrow \alpha.')^n \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ collapse\ (\nabla + \{\alpha \mapsto \beta\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ collapse\ (\nabla + \{\alpha \mapsto s \dashv (\alpha.')^n\}) \\
collapse \in \mathscr{G}(\mathcal{E}) \to \mathscr{G}(\mathcal{E}) \\
collapse\ \nabla \quad \triangleq \mathbf{if}\ \nexists \alpha : \nabla\ \alpha = s \dashv \alpha^n \wedge \{\alpha^n\} \cap \mathscr{D}(\|\nabla\|) \neq \varnothing\ \mathbf{then}\ \nabla \\
\qquad \mathbf{else\ let}\ \nabla' + \{\alpha \mapsto s \dashv \alpha^n\} = \nabla : \{\alpha^n\} \cap \mathscr{D}(\|\nabla\|) \neq \varnothing \\
\qquad\quad \mathbf{in}\ upd\ \nabla'\ \alpha\ (s \dashv \alpha^n)
\end{array}
$$

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 α in a fully collapsed graph will effectively rewrite all subterms identical to the one that can be extracted from α.

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 $(\cdots = \cdots)^n$ 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* (Turchin [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, $\mathcal{B}$). Let the set of *constructor terms* $\mathcal{B}$ be given by the grammar $b ::= x \mid c\, b^n$. The relation $\longmapsto \subseteq \mathcal{T} \times \mathcal{T}$ is defined as $\longrightarrow$ in Def. 13, except that $\mathcal{V}$ is replaced by $\mathcal{B}$ in the *const* rule. □

The new relation $\longmapsto$ 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 $\longmapsto$ we can reduce non-ground terms as long as we do not run into redices of the form $g\, x\, t^n$. 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 $\Longmapsto \subseteq \mathcal{T} \times \mathcal{T}$ is defined as $\mapsto$, but with the additional rule

$$\text{(inst)}\ \frac{\langle \nabla\, \alpha_0 \rangle \rightsquigarrow \alpha_0' \qquad \alpha_0' \notin \mathscr{D}(\nabla) \qquad (\beta_\cdot = \text{FRESH})^m \qquad \nabla' = upd\,(\nabla + \{\alpha \mapsto g \triangleleft \alpha_0\, \alpha^n\})\, \alpha_0'\, (c \triangleleft \beta^m)}{\langle (\nabla + \{\alpha \mapsto g \triangleleft \alpha_0\, \alpha^n\})\, \alpha \rangle \hookrightarrow \langle \nabla'\, \alpha \rangle}\ g\,(c\, x^m)\, y^n \stackrel{g}{=} t$$

□

The relation $\Longmapsto$ is non-deterministic. When it encounters a stuck redex $g\, x\, t^n$, it "produces" a new DAG where $g\, x\, t^n$ 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 q 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 $\underset{q \cdot I}{\Longmapsto}$ 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 $\trianglelefteq$ 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 $s_i \trianglelefteq s_j$. □

Hence, if we ensure that, for all nodes n in the process tree, there never exists an ancestor a of n such that $\text{label}(a) \trianglelefteq \text{label}(n)$, 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* $\trianglelefteq \subseteq \mathcal{T} \times \mathcal{T}$ is the smallest relation satisfying the inference rules

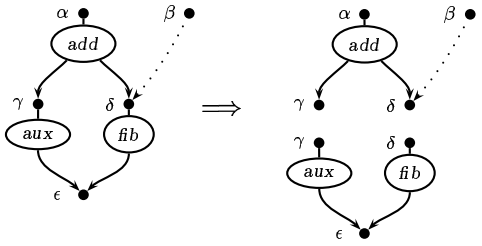
$$\frac{}{x \trianglelefteq y} \qquad \frac{(t. \trianglelefteq u.)^n}{s\ t^n \trianglelefteq s\ u^n} \qquad \frac{t \trianglelefteq u_i}{t \trianglelefteq s\ u^n}\ 1 \leq i \leq n$$ □

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.



$$\begin{aligned}
&\textbf{data}\ \mathrm{Pair}\ x\ y &&= \langle x\ y\rangle\\
&h\ x &&= g\ (f\ x)\\
&g\ \langle u\ v\rangle &&= \langle (add\ u\ v)\ v\rangle\\
&f\ x &&= \langle (aux\ x)\ (fib\ x)\rangle
\end{aligned}$$

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 ∇ is implicitly accompanied a finite *root list*, $\mathrm{roots}(\nabla) \in \mathcal{A}^*$.
2. The *ports* of a DAG ∇ is the set of addresses outside the domain of ∇ that are reachable through the roots of ∇:
$$\mathrm{ports}(\nabla) \triangleq \{\beta \in \mathcal{A} \mid \beta \notin \mathscr{D}(\nabla) \wedge \exists \alpha \in \mathrm{roots}(\nabla) : \alpha \xrightarrow{*} \beta\}\ .$$
3. A DAG ∇' is a *subdag of* a DAG ∇, denoted $\nabla' \leq \nabla$, if $\nabla' \subseteq \nabla$ and
$$\forall \alpha \in \mathscr{D}(\nabla) \setminus \mathscr{D}(\nabla') : (\alpha \multimap \beta \text{ implies } \beta \notin \{\beta \mid \alpha \in \mathrm{roots}(\nabla') \xrightarrow{+} \beta\})\ .$$
4. The pair $\langle \nabla'\ \nabla''\rangle$ is a *proper split of* ∇ if $\nabla = \nabla' + \nabla''$, $\nabla'' \leq \nabla$, and $\nabla' \neq \varnothing \neq \nabla''$. □

Informally, all nodes *external* to a subdag can only reach nodes *in* the subdag through the *roots* of the subdag. That is, if $\nabla' \leq \nabla$, then ∇' can be "carved" out of ∇, such that $\nabla = \nabla' + \nabla''$ and ∇'' interacts with ∇' only through the roots of ∇'. A proper split is then a division of a DAG into two non-trivial parts. The split in Ex. 25 is proper.

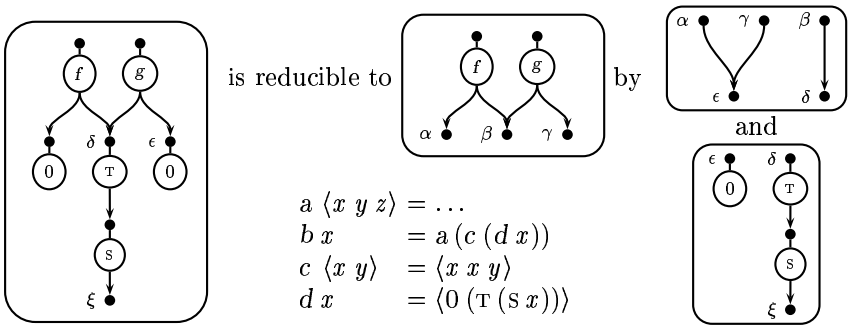
We will, however, use such a split operation as a last resort. A more sophisticated generalisation can be achieved when the offending node n in the process tree has an ancestor a such that n is *reducible to* a. Informally, ∇ is reducible to ∇_1, if the terms in ∇ can be reconstructed by carving out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

Definition 27 (terms, reducible).

1. $\mathrm{terms}(\nabla) \triangleq [(xtract\ \langle \nabla\ \alpha\rangle) \mid \alpha \leftarrow \mathrm{roots}(\nabla)]$.
2. ∇ is *reducible to* ∇_1 *by* $\nabla_2 = \{(\alpha. \mapsto \beta.)^n\}$ *and* ∇_3, if
   (a) $\mathrm{ports}(\nabla_1) = \{\alpha^n\}$,

(b) $\text{roots}(\nabla_2) = [\alpha^n]$,
(c) $\text{roots}(\nabla_3) = [\beta \mid \beta \leftarrow \text{ports}(\nabla_2)]$,
(d) $\nabla_3 \leq \nabla$, and
(e) $\text{terms}(\nabla) = \text{terms}(\nabla_1 + \nabla_2 + \nabla_3)$. □

Example 28. Assume that the DAG ∇_1 in the middle is an ancestor of the DAG ∇ to the left.



$$
\begin{aligned}
a\ \langle x\ y\ z\rangle &= \ldots \\
b\ x &= a\ (c\ (d\ x)) \\
c\ \langle x\ y\rangle &= \langle x\ x\ y\rangle \\
d\ x &= \langle 0\ (\mathrm{T}\ (\mathrm{S}\ x))\rangle
\end{aligned}
$$

The state of ∇ can be generalised into a function b by calling the function a, representing state ∇_1, by providing the arguments constructed by functions c and d, representing the indirections ∇_2 and subdag ∇_3 to the right. □

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* τ is a non-empty tree labelled with DAGs. For a particular node ν in τ, the *label of* ν is denoted $\text{label}(\nu)$, and the set of ancestors is the set of proper predecessors of ν in τ, denoted $\text{anc}(\tau, n)$. The leaves of τ are denoted by $\text{leaves}(\tau)$. A node ν in τ 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. $\text{terms}(\text{label}(\nu)) \in \mathcal{B}^*$.

A process tree is *finished*, if all nodes are finished. □

That is, a node ν in a process tree is finished if ν is an interior node, if ν is a repetition, or if there are no function symbols left to drive in ν.

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

Definition 31. We say that ∇ is *embedded in* ∇', denoted $\nabla \precsim \nabla'$, if both

$$\forall t \in \text{terms}(\nabla) : \exists u \in \text{terms}(\nabla') : t \trianglelefteq u$$
$$\forall u \in \text{terms}(\nabla') : \exists t \in \text{terms}(\nabla) : t \trianglelefteq u \ .$$

We define embeddings $\nabla\ \nabla' \triangleq$

$$\{\alpha \in \mathscr{D}(\nabla') \mid \exists \beta \in \text{roots}(\nabla) : (\mathit{xtract}\ \langle \nabla\ \beta \rangle) \trianglelefteq (\mathit{xtract}\ \langle \nabla'\ \alpha \rangle)\} \ . \qquad \square$$

Conjecture 32. $\precsim$ is a well-quasi-order.

Remark 33. As of this writing, we have not been able establish a proof of the 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 τ
let $\alpha_0 = \text{FRESH}$
let tree τ consist of a single node labelled $(\mathit{subst}_I\ \varnothing\ \alpha_0\ t\ \phi^{-1})$ and roots $[\alpha_0]$
while τ is unfinished **do**
  let ν be an unfinished node with $\nabla = \text{label}(\nu)$
  if $\nexists \mu \in \text{relanc}(\tau, \nu) : \text{label}(\mu) \precsim \nabla$
  then let $\alpha \in \text{roots}(\nabla) : \exists \nabla' : \langle \nabla\ \alpha \rangle \overset{q \cdot I}{\longmapsto\!\!\!\!\Longrightarrow} \nabla'$
    add children to ν with labels $[\nabla' \mid \langle \nabla\ \alpha \rangle \overset{q \cdot I}{\longmapsto\!\!\!\!\Longrightarrow} \nabla']$
  else let $\mu \in \text{relanc}(\tau, \nu) : \nabla_1 = \text{label}(\mu) \wedge \nabla_1 \precsim \nabla$
    if ∇ is reducible to ∇_1 by ∇_2 and ∇_3
    then add three children to ν with labels ∇_1, ∇_2, and ∇_3
    else if $\exists\, \langle \nabla_2\ \nabla_3 \rangle : \langle \nabla_2\ \nabla_3 \rangle$ is a proper split of ∇ and
      $\text{ports}(\nabla_2) \cap (\text{embeddings}\ \nabla_1\ \nabla) \neq \varnothing$
    then add two children to ν with labels ∇_2 and ∇_3
    else let $\langle \nabla_2\ \nabla_3 \rangle$ be a proper split of ∇_1
      replace all subtrees of μ with two children labelled ∇_2 and ∇_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 ∇ is speculatively unfolded to a DAG ∇' by an instantiation step, it is always the case that $\nabla \precsim \nabla'$. Secondly, an instantiation step will give rise to a series of *deterministic* unfoldings (Turchin [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 ν by a node in a process tree τ.

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

$$\mathrm{relanc}(\tau, \nu) \triangleq \begin{cases} \{\mu \mid \mu \in \mathrm{anc}(\tau, \nu) \wedge \mu \text{ is global}\} & \text{if } \nu \text{ is global} \\ \mathrm{locanc}(\tau, \nu) & \text{if } \nu \text{ is local} \end{cases} \qquad \square$$

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.[2] 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.

[2] To some extent, the efficiency also depends on the treatment of sharing between the outermost constructors; the produced code must carefully mimic such sharing.

Example 37. Let q be the Fibonacci Number program in Fig. 1, let $t = \mathit{fib}\, x$. 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:

$$
\begin{array}{lcl|lcl}
\textbf{data}\ \mathrm{Nat} & = & 0 \mid \mathrm{s}\,\mathrm{Nat} & d\,(\mathrm{s}\,x) & = & f\,(d\,x) \\
\textbf{data}\ \mathrm{Pair}\ x\ y & = & \langle x\ y\rangle & e\,0\,y & = & y \\
a\,0 & = & \mathrm{s}\,0 & e\,(\mathrm{s}\,x)\ y & = & \mathrm{s}\,(e\ x\ y) \\
a\,(\mathrm{s}\,x) & = & b\,x & f\ \langle x\ y\rangle & = & g\ x\ y \\
b\,0 & = & \mathrm{s}\,0 & g\,0\,y & = & \langle y\,0\rangle \\
b\,(\mathrm{s}\,y) & = & c\,(d\,y) & g\,(\mathrm{s}\,x)\ y & = & h\,(i\,(g\ x\ y)) \\
c\ \langle x\ y\rangle & = & e\ x\ y & h\ \langle x\ y\rangle & = & \langle(\mathrm{s}\,y)\ x\rangle \\
d\,0 & = & \langle(\mathrm{s}\,0)\ (\mathrm{s}\,0)\rangle & i\ \langle x\ y\rangle & = & \langle(\mathrm{s}\,y)\ x\rangle
\end{array}
$$

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. □

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 λ-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 λ-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 Søndergaard [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

1. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machines*, 24(1):44–67, 1977.
2. Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, 14-16 June 1993. ACM Press.
3. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.
4. J. Gallagher. Tutorial in specialisation of logic programs. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[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.

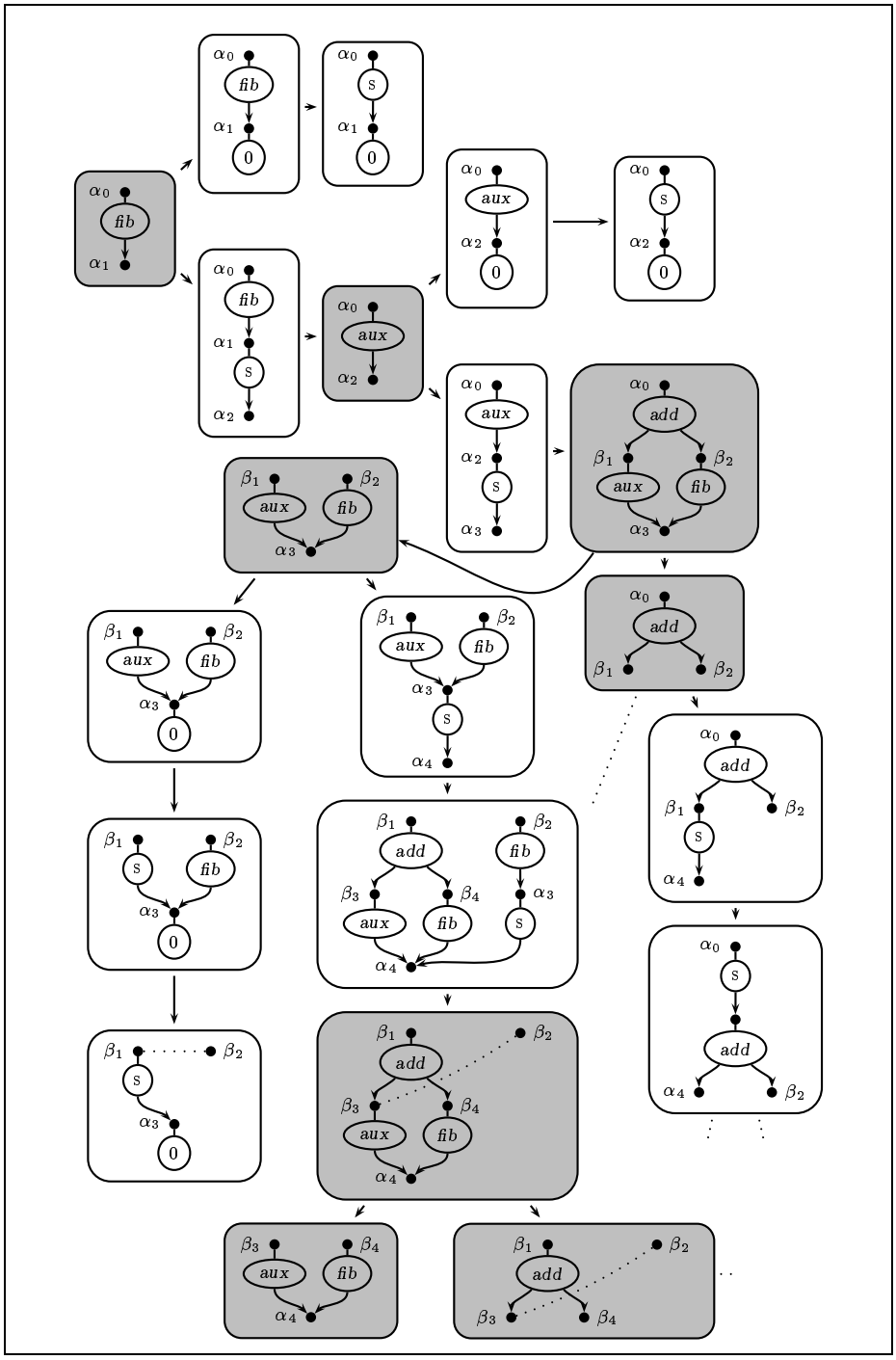


Fig. 4. Process tree of the Fibonacci program. Global nodes are shaded.

5. R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
6. Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. *Fundamenta Informaticae*, XV:37–60, 1991.
7. Berthold Hoffmann and Detlef Plump. Implementing term rewriting by jungle evaluation. *RAIRO Theoretical Informatics and Applications*, 25(5):445–472, 1991.
8. N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
9. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
10. Michael Leuschel. On the power of homeomorphic embedding for online termination. In Georgio Levi, editor, *Static Analysis. Proceedings*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
11. A. Pettorossi, E. Pietropoli, and M. Proietti. The use of the tupling strategy in the development of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 111–151. Kluwer Academic Publishers, 1993.
12. Alberto Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 273–281. ACM, ACM, August 1984.
13. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.
14. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
15. M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
16. Morten Heine Sørensen and Robert Glück. Introduction to supercompilation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.
17. Morten Heine B. Sørensen. Convergence of program transformers in the metric space of trees. *Science of Computer Programming*, 37(1–3):163–205, May 2000.
18. V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549, Amsterdam: North-Holland, 1988. Elsevier Science Publishers B.V.
19. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
20. V.F. Turchin and A.P. Nemytykh. A self-applicable supercompiler. Technical Report CSc. TR 95-010, City College of the City University of New York, 1995.
21. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.
22. Christopher Peter Wadsworth. *Semantics and pragmatics of the lambda calculus.* Ph.D. thesis, Programming Research Group, Oxford University, September 1971.