Supercompilation: Ideas and Methods

by Ilya Klyuchnikov ⟨ilya.klyuchnikov@gmail.com⟩
and Dimitur Krustev ⟨dkrustev@gmail.com⟩

*Supercompilation (**supervised compilation**) is a program transformation technique based on the construction of a **full and self-contained model** of the program.*

*We illustrate the fundamental ideas and methods of supercompilation with a working supercompiler called **SC Mini** [1, 2] for a very simple purely functional language.*

The Essence of Supercompilation

Supercompilation was invented by V. F. Turchin in the Soviet Union during the 1970s. In his own words [3]:

> A program is seen as a machine. To make sense of it, one must observe its operation. So a supercompiler does not transform the program by steps; it controls and observes (SUPERvises) the machine, let us call it M_1, which is represented by the program. In observing the operation of M_1, the supercompiler COMPILES a program which describes the activities of M_1, but it makes shortcuts and whatever clever tricks it knows, in order to produce the same effect as M_1, but faster. The goal of the supercompiler is to make the definition of this program (machine) M_2 self-sufficient. When this is achieved, it outputs M_2 in some intermediate language L^{sup} and simply throws away the (unchanged) machine M_1...
>
> A supercompiler would run M_1 in a general form, with unknown values of variables, and create a graph of states and transitions between

> possible configurations of the computing system ... in terms of which the behavior of the system can be expressed. Thus the new program becomes a self-sufficient model of the old one.

Article Goals

Functional programming practitioners, who are aware of supercompilation, have various opinions. The most common reaction is one of mistrust or even full rejection, mostly because of the fact that there is no industrial-strength supercompiler yet. All existing supercompilers have an experimental status and, in most cases, are only successfully used by their own authors.

Supercompilation is, in a certain sense, a very general method, but, in practice, only specialized tools – re-using just some of the ideas of supercompilation – work well enough to be useful. This clashes with the expectation that a general tool should come out of a general idea. Another possible reason for the mistrust is the fact that supercompilation is still lacking a "killer application".

One common misconception persists: that supercompilation is only a tool for program optimization. Supercompilation is a **program transformation technique**. Program transformations can have different goals. One such goal can be optimization, another – program analysis. Supercompilation is equally applicable to both these goals. Most works dealing with supercompilation consider only the optimization aspect, but we should not be misled that optimization is the only useful application of supercompilation.

We must distinguish the notions of **supercompilation** and **a supercompiler**. Many articles describe various technical difficulties arising from the implementation of a specific supercompiler and ways to overcome these difficulties, while mostly leaving aside the underlying key ideas of supercompilation. In most cases, the parts of a given supercompiler interact in intricate ways, which may give the impression of a complicated monolithic construction. Or, as Simon Peyton-Jones put it in the interview [4]: "... To build a supercompiler, if you look at how it works, there are a number of things all wound together in one rather complicated ball of mud. I found it extremely difficult when I really wanted to understand it. I found it very difficult to understand the papers. ...".

The main goal of this article is to give a clear and concise illustration of the aforementioned Turchin's quote, describing the essence of supercompilation by using a minimalistic example supercompiler. The accent is on delineating and explaining the basic building blocks and showing how these blocks can be implemented and made to work together in a clear and simple fashion.

Organization of the article: A important companion to this article is the full source code of the toy supercompiler SC Mini, with detailed comments [1] (250

lines of main code + 200 lines of auxiliary utilities in Haskell). The text of the article itself can be seen as an introduction to studying these supercompiler sources. The article introduces and explains the main supercompiler terminology and ingredients, and we illustrate SC Mini's behavior on suitable examples. These examples are chosen to be neither too complicated, nor oversimplified. The reader is thus invited to carefully study all examples.

Things left out: Do not expect to find a detailed comparison of supercompilation to other methods of program optimization/transformation – it would require a survey article of a much larger scope. A comprehensive bibliography of supercompilation literature is similarly out of scope, although we do give many useful references where relevant.

Supercompilation by Example

> The advantages of every formal language are determined not only by its ease of use by humans, but also by the amenability of its texts to formal transformations.
>
> V.F. Turchin [5]

It would be nice if all computer-science articles were accompanied by some formal executable description (in the form of a program or a formal specification) in order to ensure reproducibility of results.

The SC Mini supercompiler is such an executable description of the fundamental methods of supercompilation, which are the main topic of this article. SC Mini is based on the supercompiler described in the groundbreaking MSc thesis of Morten H. Sørensen [6] (which remained only on paper).

No supercompilation description is complete without describing subtle fundamental notions such as program semantics, evaluation results, computation state. All these notions are formally represented in the sources of SC Mini, which helps avoid ambiguities.

SC Mini transforms programs written in a toy language called SLL (= Simple Lazy Language; corresponding roughly to Sørensen's M_0). Paraphrasing Turchin's quote from the beginning of the section, we argue that SLL's main advantages are:

1. A simple language definition, and
2. A simple definition of a supercompiler for SLL programs.

Despite the fact that SC Mini is a supercompiler for a specific language, the methods used in its construction are fairly general and appear with only small variations in most supercompilers.

P	::=	$d_1 \ldots d_n$	program
d	::=	$f(v_1, \ldots, v_n) = e;$	"indifferent" function
	|	$g(p_1, v_1, \ldots, v_n) = e_1;$	"curious" function
		$\ldots$	
		$g(p_m, v_1, \ldots, v_n) = e_m;$	
e	::=		expression
		v	variable
	|	$C(e_1, \ldots, e_n)$	constructor
	|	$f(e_1, \ldots, e_n)$	function call
p	::=	$C(v_1, \ldots, v_n)$	pattern

Figure 1: SLL abstract syntax

We shall see further through several examples that SC Mini can optimize programs (giving first a formal definition of a program optimizer), and it can also be used to automatically prove different program properties.

SLL Object Language

Defining a language entails defining its syntax and semantics. Syntax is typically defined using a context-free grammar. One possible way to define semantics is to implement an interpreter for the language. The following natural-language description is more succinctly and simply re-expressed – using Haskell – in SC Mini's sources.

Fig. 1 gives the abstract syntax of SLL. An SLL expression can be one of the following:

- variable,
- constructor, having SLL expressions as arguments, or a
- function call, having SLL expressions as arguments.

We assume further the same convention as in Haskell: constructor names start with an uppercase letter; variable names start with a lowercase letter.

An SLL expression consisting only of constructors is called a **value**. An SLL expression without any variables inside is called a **closed expression**. We define **configuration** as an alias of an SLL expression with free variables.

An SLL program consists of function definitions. We distinguish two kinds of functions – "indifferent" and "curious" (originally called by Sørensen in [6] f- and g-functions respectively). Curious functions replace case expressions (which are

```
add(Z(), y)  = y;
add(S(x), y) = S(add(x), y);

mult(Z(), y)  = Z();
mult(S(x), y) = add(y, mult(x, y));

sqr(x) = mult(x, x);

even(Z())  = True();
even(S(x)) = odd(x);

odd(Z())  = False();
odd(S(x)) = even(x);

add'(Z(), y)  = y;
add'(S(v), y) = add'(v, S(y));
```

Figure 2: `prog1`: Working with Peano numbers

always lifted to the top level). Indifferent functions just transfer their arguments to other functions or constructors; their definition is just a single expression. Curious functions perform a simple pattern match on their first argument; their definitions consist of many expressions – one per case.

SLL has no built-in data types (Booleans, numbers, etc.). SLL can be straightforwardly extended with Haskell-like data-type declarations and Hindley-Milner type inference, but we ignore issues of typing from now on, silently assuming that all programs under consideration would be well-typed under such a type system. We assume the following convention:

- the constructors `True()` and `False()` represent Boolean values,
- the value `Z()` represents 0, `S(Z())` - 1, `S(S(Z()))` - 2, etc. If the value `n` corresponds to the natural number n, then the value `S(n)` corresponds to the natural number $n + 1$ (the so-called Peano numbers).

Fig. 2 shows a program defining some functions for working with Peano numbers. It contains a single indifferent function – squaring (`sqr`). All the other functions are curious, as they do a pattern match on their first argument.

A **substitution** binds variables $v_1, v_2, \ldots, v_n$ to expressions $e_1, e_2, \ldots, e_n$ and is written as a list of pairs $\{v_1 := e_1, \ldots, v_n := e_n\}$. Application of a substitution to an expression e is defined in the usual way, and is denoted $e/\{v_1 := e, \ldots, v_n := e_n\}$.

Exercise 1. In SC Mini's sources substitution is literally represented as a list of pairs. Application of a substitution `s` to an expression `e` is denoted as `e // s`.

$$
\begin{array}{lll}
\mathcal{I}_p[\![e]\!] & \Rightarrow e & (I_1) \\
 & \text{if } e \text{ is a value} & \\
\mathcal{I}_p[\![C(e_1,\ldots,e_n)]\!] & \Rightarrow C(\mathcal{I}_p[\![e_1]\!],\ldots,\mathcal{I}_p[\![e_n]\!]) & (I_2) \\
 & & \\
\mathcal{I}_p[\![con\langle f(e_1,\ldots,e_n)\rangle]\!] & \Rightarrow \mathcal{I}_p[\![con\langle e/\{v_1:=e_1,\ldots,v_n:=e_n\}\rangle]\!] & (I_3) \\
 & \text{if } f(v_1,\ldots,v_n) \stackrel{\text{p}}{=} e & \\
\mathcal{I}_p[\![con\langle g(C(e_1,\ldots,e_m),e_{m+1},\ldots,e_n)\rangle]\!] & \Rightarrow \mathcal{I}_p[\![con\langle e/\{v_1:=e_1,\ldots,v_n:=e_n\}\rangle]\!] & (I_4) \\
 & \text{if } g(C(v_1,\ldots,v_m),v_{m+1},\ldots,v_n) \stackrel{\text{p}}{=} e &
\end{array}
$$

Figure 3: SLL: interpreter $\mathcal{I}_p$ for a program p

con	::=	$\langle\rangle \mid g(con,\ldots)$	context
red	::=	$f(e_1,\ldots,e_n) \mid g(C(e_1,\ldots,e_n),\ldots)$	redex

Figure 4: SLL: context and redex

Find `e`, `v1`, `e1`, `v2`, `e2`, such that
`e // [(v1, e1), (v2, e2)]` $\neq$ `e // [(v1, e1)] // [(v2, e2)]`.

Fig. 3 shows the formal reduction semantics of SLL expressions. The notation $e_1 \stackrel{\text{p}}{=} e_2$ means that the program p contains a definition $e_1 = e_2$. The semantics is defined through a rewriting SLL interpreter with normal-order reduction. The SLL interpreter $\mathcal{I}_p$ processing a program p evaluates, step by step, each closed SLL expression to an SLL value (or falls into an infinite loop, or terminates with an error message, but we shall often ignore the last case for simplicity). As far as reduction is concerned, we distinguish two kinds of closed expressions:

1. $e = C(e_1,\ldots,c_n)$ – a constructor is "pushed" outside, and we proceed to reduce its arguments.
2. $e \neq C(e_1,\ldots,c_n)$ – then we locate the leftmost reducible function call (**redex**, see Fig. 4) and unfold it according to the corresponding definition from the program. A function call is reducible if it is a call to 1) an indifferent function or 2) a curious function, where the first argument starts with a constructor.

The rules for evaluating SLL expressions are (as typical for functional languages) **compositional**, meaning that, if the expression $e_1/\{v := e_2\}$ is closed, then (taking laziness into account):

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

Remark that we can consider $\mathcal{I}_p$ as a machine, describing the program p.

Exercise 2. In SC Mini the interpreter is defined as a function `eval`. The notation $\mathcal{I}_p[\![e]\!]$ corresponds to the call `eval p e`. Prove that the evaluation of

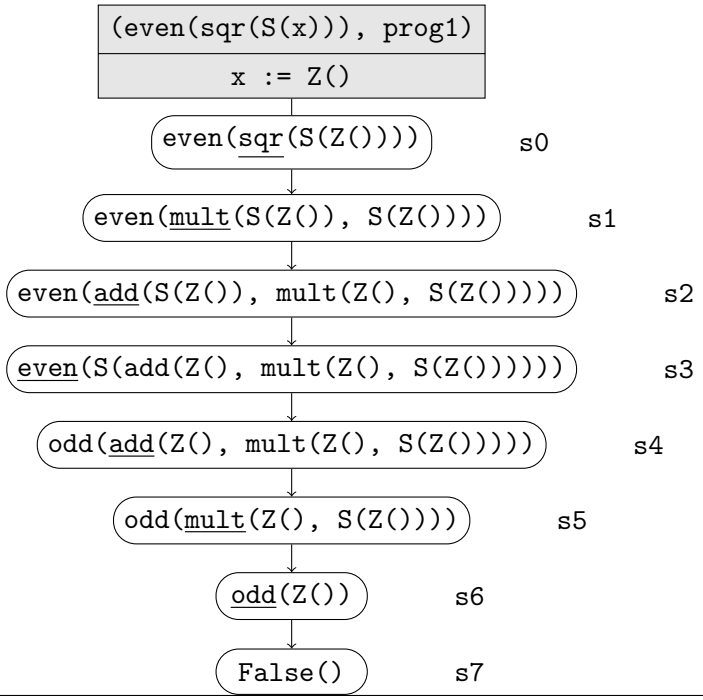


Figure 5: An example of a task evaluation

`eval p (e1 // [(v, e2)]) == eval p (e1 // [(v, eval p e2)])` cannot give `False`.

We call an **SLL-task** (or sometimes simply **task**) the pair (e, p) of a configuration e and a program p (recall that a configuration is simply an expression with free variables). The evaluation of a task (e, p) on arguments $args$ is defined as:

$$\mathcal{R}_{(e,p)}[\![args]\!] = \mathcal{I}_p[\![e/args]\!]$$

(In SC Mini the corresponding call is `sll_run (e, p) args`.)

Fig. 5 shows an example of evaluating the expression `even(sqr(S(Z())))` with the interpreter for the program `prog1`. The redexes are underlined. The root node represents the task with its arguments.

Next, we formally define what is an **optimizer** of SLL tasks. An SLL optimizer takes a task (e, p) as input and outputs another task (e', p'). Our first requirement is correctness: for all arguments $args$, the results of both tasks must be the same:

$$\mathcal{R}_{(e,p)}[\![args]\!] = \mathcal{R}_{(e',p')}[\![args]\!]$$

Our second requirement is optimization: the evaluation of the second task must require no more reduction steps than the evaluation of the first one. (The number

of steps of a reduction sequence can be computed by the function `sll_trace` in SC Mini's sources.)

An Overview of SC Mini

SC Mini transforms a task (e, p) into another task (e', p'), proceeding as follows:

1. It constructs a "machine" $\mathcal{M}_p$, which encodes the behavior of the interpreter $\mathcal{I}_p$ of a program p over a generalized input – configurations (instead of values);
2. It performs a finite (but large enough) number of executions of the machine $\mathcal{M}_p$, analyzing their results;
3. It builds a finite "graph of configurations", which fully describes the behavior of $\mathcal{M}_p$ during those executions; and
4. It simplifies the graph of configurations and builds a new task (e', p') from it.

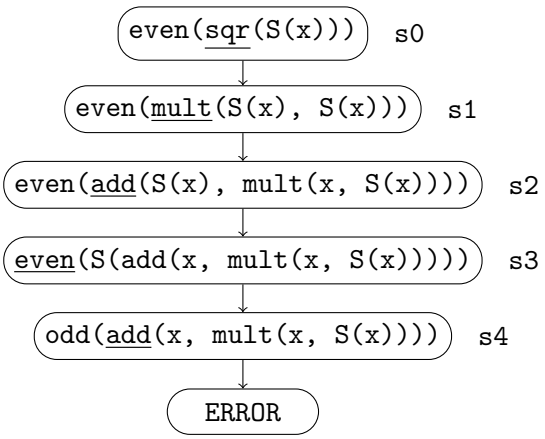
We now explain these steps in detail.

Driving

What would happen, if we tried to perform the following task on an empty list of arguments?

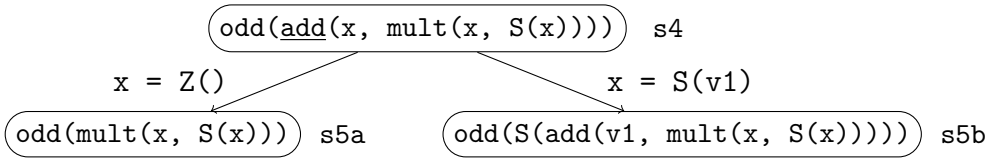
```
(even(sqr(S(x))), prog1)
```

The SLL interpreter is defined in such a way that it can actually evaluate (to some degree) expressions with free variables:

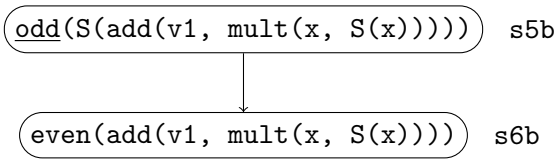


The interpreter does not expect a variable as a first argument of `add`. The presence of such a free variable `x`, however, does not stop it from performing some initial reduction steps, mostly thanks to the call-by-name reduction strategy.

Let us "extend" the interpretation process $\mathcal{I}$: instead of stopping with an error, let us consider the different ways in which evaluation might proceed. The definition of `add` in `prog1` provides 2 possibilities: either `x = Z()`, or `x = S(x1)`:



Now evaluation can proceed with the right subexpression, and so on:



The interpreter $\mathcal{I}_p$ is intended to reduce step-by-step closed expressions. The result of interpretation $\mathcal{I}_p[\![e]\!]$ is some SLL value (unless there is an infinite loop). We introduce a machine $\mathcal{M}_p$, which will compute over configurations. Each transition of this machine – $\mathcal{M}_p[\![c]\!]$ – models a step of the interpreter; it can produce one or several new configurations, labeled with the kind of step taken. We distinguish the following kinds of machine steps:

1. Transient step – the corresponding interpretation step does not depend on the value of any free variable in the configuration. Example: the move from state `s0` to state `s1`.
2. Stop – further modeling is not possible. This occurs when the expression to be reduced is a variable or a value.
3. Decomposition – some part of the result is already known. Example: in the expression `S(sqr(x))`, the outer constructor is clearly a part of the result. We can continue processing its subexpressions.
4. Case analysis – modeling cannot proceed unambiguously. We can, however, consider **all possible further steps of the interpreter, as defined in the program**. Example: the transitions from state `s4` to states `s5a` and `s5b`.

Such modeling – which is a key part of supercompilation – is called **driving**. The preceding paragraph informally defines one driving step.

Exercise 3. In the SC Mini sources building the machine $\mathcal{M}_p$ is performed by `driveMachine p`. Show that `driveMachine p` is powerful enough to evaluate closed expressions. In other words, define a function `f`, such that
`f (driveMachine p) e = eval p e`

Trees of Configurations

The **tree of configurations** for an SLL task (e, p) is built in the following way. We create a machine $\mathcal{M}_p$ modeling p. In the beginning, the tree starts as a single node, labeled with the start configuration e (the task goal). Then, for each tree leaf n, which contains a configuration c, we perform one machine transition $\mathcal{M}_p[\![c]\!]$ and attach new children nodes to n, each of them containing one of the configurations resulting from this transition. We repeat the process for each new leaf. The resulting tree is called "tree of configurations", as each node is labeled by a configuration. Note that the trees of configurations are sometimes called "process trees" in the literature.

Of course, the tree built in this way will be infinite in general. But we can always consider the tree only up to a given (finite) depth (in the previous subsection we have already seen an example – the top of the tree arising from the task `(even(sqr(S(x))), prog1)`).

Exercise 4. SC Mini builds the tree of configurations for a task (e, p) as follows:
`buildTree (driveMachine p) e`
Try to find a characterization of the class of tasks, which result in finite trees of configurations.

The notion of a tree of configurations is not strictly necessary for building a working supercompiler. Many existing supercompilers, especially those built in recent years, make no explicit use of either trees of configurations or graphs of configurations (which will be introduced shortly). Trees and graphs of configurations can be useful, however, both from theoretical and from educational point of view, as they open the way for defining a much more modular supercompiler.

Let's assume for a moment, that we can build (and somehow store) infinite trees of configurations. Then, if for a task (e, p), using a machine $\mathcal{M}_p$, we have built a tree of configurations t, we can throw away the program p and the machine $\mathcal{M}_p$, and only work with the tree t. This is possible, as the tree t fully describes the computation of the task (e, p) without any reference to the text of the program p.

Exercise 5. SC Mini sources contain an executable evidence for the last statement – a "tree interpreter" `intTree`, which can perform tasks using only the corresponding tree of configurations, and not the original program. If `t` is the tree of configurations for the task `(e, p)`, then `intTree t args` gives the result of computing the task over arguments `args`. Convince yourselves that
`eval (e, p) args == intTree (buildTree (driveMachine p) e) args`
cannot produce `False`.

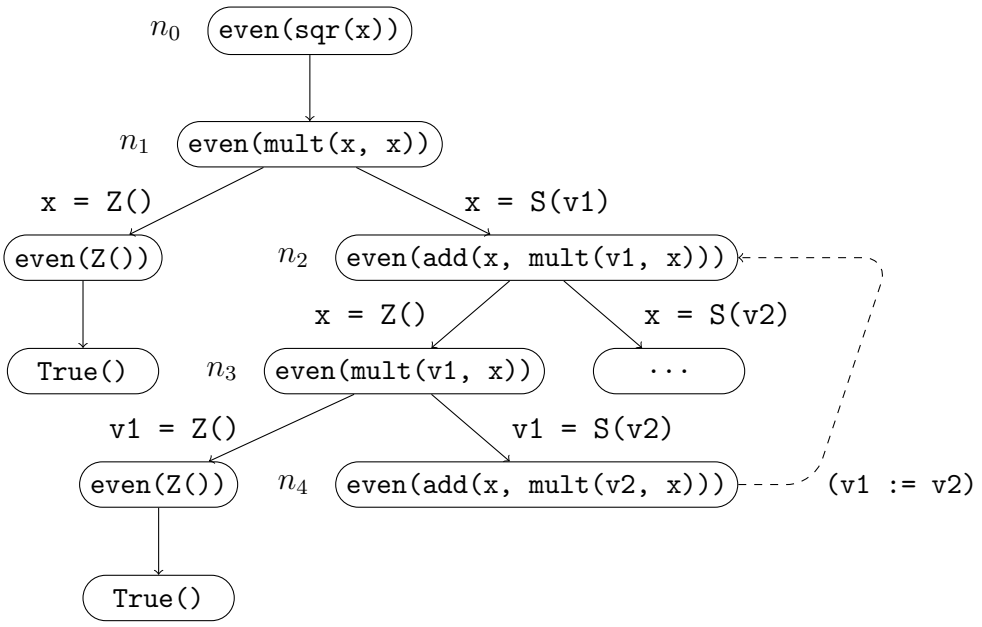
Folding

> Infinite driving graphs are useful in many ways, but to use a graph as an executable program it must be finite.
>
> V. F. Turchin [7]

We can evaluate `even(sqr(S(x)))` for each `x`, by building the tree of configurations and then using the "tree interpreter" `intTree`. The tree of configurations is, however, infinite. The goal of **folding** is to turn the infinite tree into a finite object, from which we could recover the original infinite tree, if needed.

How does folding work? Assume that, while building the tree of configurations, there is a path with nodes $n_0 \to \ldots \to n_i \to \ldots \to n_j \to \ldots$ Assume further that the configuration in node n_j is a renaming (differs only in the choice of variable names) of the configuration in another node n_i $(i < j)$. It is clear that we can reconstruct the subtree starting from node n_j by the subtree starting from node n_i if we systematically rename the variables in the corresponding configurations.

Consider the tree of configurations for the task `(even(sqr(x)), prog1)`:



The configuration in node n_4 is a renaming of the one in node n_2. The (infinite) subtrees starting from node n_2 and from n_4 differ only by the names of the variables in the corresponding nodes. So, we can simply memorize that the subtree from n_4 can be built by a given renaming from the subtree starting at n_2, without building explicitly this subtree. It is important to note that the reconstruction of

the subtree n_4 from the subtree n_2 does not require the original program (nor the machine $\mathcal{M}_p$).

We denote such situations with a special kind of edge, leading from the lower node to the upper one. As a result, the tree of configurations is no longer a tree but rather a **graph of configurations** (also sometimes called "process graph").

We are lucky with the tree of configurations for the task `(even(sqr(x)), prog1)` – it folds into a graph. We can see, that in such a way we can turn some infinite trees into finite graphs. The resulting graph will serve as a new self-contained representation of the given task. We shall denote the graph for the task t as $\mathcal{G}_t$.

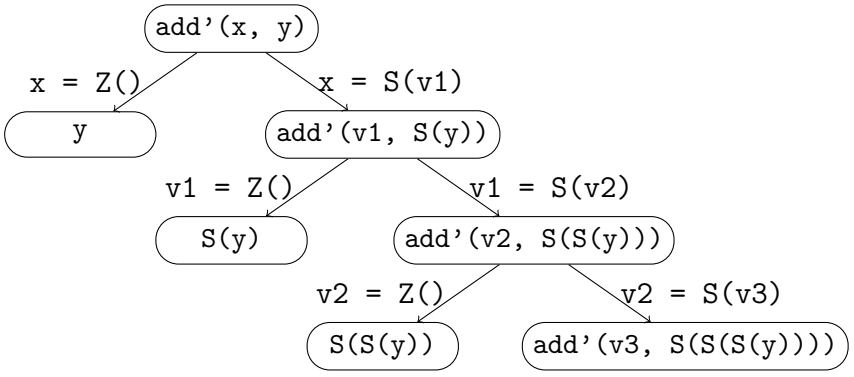
Exercise 6. Just some small modifications of the tree interpreter `intTree` are sufficient to make it work also with folded graphs of configurations. Take a look in SC Mini's sources to see how it is done.

The graph $\mathcal{G}_t$ for the task $t = (e, p)$ can be converted into a new task (e', p'). The resulting program `p'` is called **residual**, and we shall also call the corresponding task $(e',\ p')$ residual.

Exercise 7. SC Mini's sources contain a function `residuate`: `residuate g` transforms a graph of configurations `g` into a new task `(e', p')`.

Generalization: converting a tree into a foldable tree

We were lucky with the example in the previous subsection, as it was possible to fold the tree into a graph. It would be splendid if all trees of configurations were foldable! Unfortunately, this is not the case in general. The program `prog1` contains a function `add'`, which defines addition using an accumulating parameter. The construction of the tree of configurations for `(add'(x, y), prog1)` goes as follows:



...

This tree is not folding. Indeed, a configuration can only be a renaming of another if both have the same size. In this example, the path through the rightmost branches of the tree contains ever-growing configurations. However, if the size of configurations be bounded, the tree will always fold to a finite graph.

Exercise 8. Try to prove the last statement. *Hint:* To what kind of relation does renaming naturally give rise?

We can thus apply a divide-and-conquer heuristics. We limit the size of configurations inside nodes by some constant `sizeBound`, and if the size of a configuration becomes larger than this constant, we split it into smaller parts which we can process **independently**.

Recall that the evaluation of closed SLL expressions is compositional. For a closed expression $e_1/\{v := e_2\}$ we have:

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

The same property of compositionality holds for evaluation of tree of configurations. (Look into SC Mini source code to see how `intTree` handles splitted configurations.)

SC Mini limits the size of configurations in the tree and builds a foldable tree in the following way:

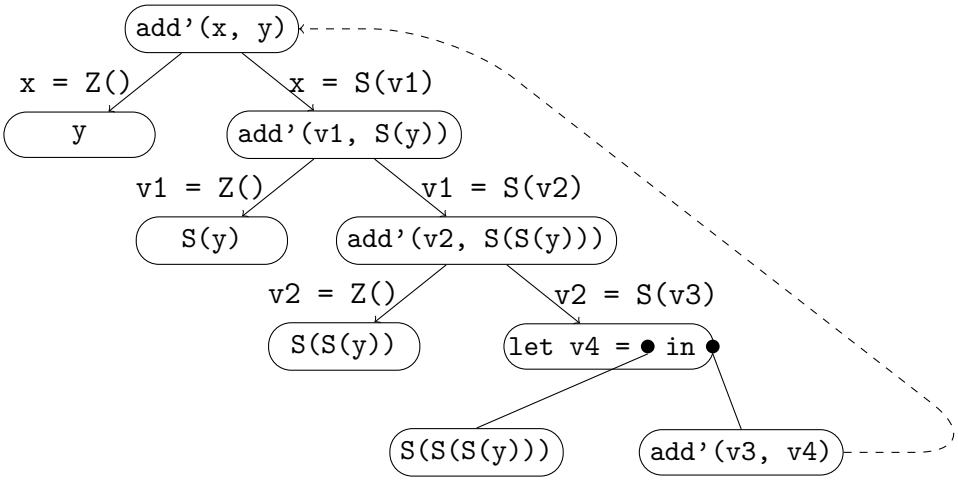
- if while building the tree a configuration e is encountered, which is a function call, and whose size is greater than `sizeBound`, then this configuration is splitted,
- otherwise, the construction of the tree is done in the standard way.

Splitting represents the configuration e as $e = e_1/(v := e_2)$, where e_2 is the largest subexpression of e, and e_1 – the original expression e, in which e_2 is replaced by a (fresh) variable v. Such configurations are encoded as `let`-expressions $let\ v = e_2\ in\ e_1$. Then we process the configurations e_1 and e_2 independently.

Exercise 9. We do not need to check the size of constructor nodes. Why?

The configuration e_1 is a generalization of e. And e is, in turn, a special instance of e_1. By a slight abuse of terminology, the process of transforming the tree of configurations into a foldable one – as described above – is also called **generalization**.

Now, if we limit the growth of configurations (by size), for any task we can build a (potentially infinite) tree of configurations, which will in all cases be folded into a **finite** graph of configurations. Returning to our current example, if we set `sizeBound` equal to 5, then as a result of generalizing the rightmost configuration we obtain:



Exercise 10. What properties should a task have, in order for the configuration tree to be foldable without using generalization?

Exercise 11. It is very easy to extend `intTree` with treatment of `let`-expressions. Check how it is done in the SC Mini sources.

The idea of generalizing configurations is one of the cornerstones of supercompilation. What we have described above is one of the possibly simplest and least sophisticated ways to perform generalization.

The supercompiler part responsible for deciding when to generalize is historically called the **whistle**. Whistles in most existing supercompilers are typically more complicated than just limiting configuration size.

The term "whistle", despite sounding a bit unscientific, is generally accepted and used in most descriptions of supercompilation. The whistle is a heuristic component, whose task is to signal the danger of possibly non-foldable (and thus infinite) fragments appearing in the tree of configurations. Of course, this task is an instance of the halting problem (if the object language is Turing-complete) and does not have an exact solution. Any whistle will necessarily give only approximative results, and even the best heuristics cannot guarantee the absence of blunders, such as whistling just a few steps before an actually foldable configuration. The only strict requirement is to always err on the side of caution and never let infinite tree paths slip through, as it would make the supercompiler itself loop.

Supercompiler prototype

In the SC Mini sources generalization and folding are implemented via following functions:

- `buildFTree` – it builds a foldable tree of configurations by limiting size of configurations (by means of splitting)

- `foldTree` – it takes a (possibly infinite) tree of configurations and folds it into a finite graph.

Let's create the following program transformer:

```
transform :: Task -> Task
transform (e, p) =
        residuate $ foldTree $ buildFTree (driveMachine p) e
```

A most interesting thing in this transformer happens there: the machine $\mathcal{M}_p$ is not only run – the result of its run is analyzed, and possibly a decision to generalize is taken.

We argue that this program transformer fulfills the requirements of the previously stated definition of "supercompiler". Although this transformer performs no interesting optimizations, it can serve as a foundation for a full-blown supercompiler. We shall use it as a baseline to which we shall compare the transformers that follow (deforestation and supercompilation).

Exercise 12. Try to show that for each task `(e, p)` and each substitution `s` the computation `sll_run (e', p') s` (where `(e', p')` is the corresponding residual task) requires exactly as many reduction steps as the computation `sll_run (e, p) s`. That means that the transformer `transform` does not degrade performance, but it does not improve it either.

Does this transformer modify the input program at all? Yes, and here is an example:

`(even(sqr(x)), prog1)` $\overset{transform}{\Longrightarrow}$ `(f1(x), prog1T)`

Where `prog1T =`

```
f1(x) = g2(x, x);
g2(Z(), x) = f3();
g2(S(v1), x) = g4(x, x, v1);
f3() = True();
g4(Z(), x, v1) = g5(v1, x);
g4(S(v2), x, v1) = f7(v2, v1, x);
g5(Z(), x) = f6();
g5(S(v2), x) = g4(x, x, v2);
f6() = True();
f7(v2, v1, x) = g8(v2, v1, x);
g8(Z(), v1, x) = g9(v1, x);
g8(S(v3), v1, x) = f16(v3, v1, x);
g9(Z(), x) = f10();
g9(S(v3), x) = g11(x, x, v3);
f10() = False();
g11(Z(), x, v3) = g9(v3, x);
```

```
g11(S(v4), x, v3) = f12(v4, v3, x);
f12(v4, v3, x) = g13(v4, v3, x);
g13(Z(), v3, x) = g14(v3, x);
g13(S(v5), v3, x) = f7(v5, v3, x);
g14(Z(), x) = f15();
g14(S(v5), x) = g4(x, x, v5);
f15() = True();
f16(v3, v1, x) = g17(v3, v1, x);
g17(Z(), v1, x) = g18(v1, x);
g17(S(v4), v1, x) = f7(v4, v1, x);
g18(Z(), x) = f19();
g18(S(v4), x) = g4(x, x, v4);
f19() = True();
```

We can easily detect at least one property of the transformer – the resulting program contains no nested function calls: it is "flat".

KMP Test

Consider the program in Fig. 6 (where function names are deliberately short, to keep the drawings of graphs of configurations small): the function `match(p, s)` checks if a string `p` is contained inside the string `s`. For simplicity we use a 2-letter alphabet – `'A'` and `'B'`. Strings are represented – as in Haskell – by lists of characters. We also use some list/string syntactic sugar from Haskell for readability, even if it is not actually implemented in SLL.

The function `match` is general, but inefficient. Consider the call `match("AAB", s)`, which checks if the substring (pattern) `"AAB"` appears inside the string `s`. Our program will perform this check in the following way: compare `'A'` with the first character of `s`, `'A'` – with the second one, `'B'` – with the third one. If any of these comparisons fails, we restart the comparison sequence after skipping the first character of `s`. This strategy is far from optimal, however. Let's assume that the string `s` starts with `"AAA..."`. The first 2 comparisons will succeed, the 3rd one will fail. It is inefficient to repeat the same sequence of comparisons on the tail `"AA..."`, as we already have enough information to know, that the first 2 comparisons of `"AAB"` with `"AA..."` will succeed. The deterministic finite automaton (DFA), built by the Knuth-Morris-Pratt algorithm [8], will consider each character of `s` at most once.

A simple way to estimate the power of an optimizing transformation is to see if it can produce – fully automatically – a well-known efficient algorithm from a "naïve", less-efficient one. In the case of supercompilation, it turns out that – starting from a naïve string-matching algorithm, and fixing the value of the substring we try to match – we can automatically obtain an efficient specialized matching program,

```
match(p, s) = m(p, s, p, s);

-- matching routine
-- current pattern is empty, so match succeeds
m("", ss, op, os) = True();
-- proceed to match first symbol of pattern
m(p:pp, ss, op, os) = x(ss, p, pp, op, os);

-- matching of the first symbol
-- current string is empty, so match fails
x("", p, pp, op, os) = False();
-- compare first symbol of pattern with first symbol of string
x(s:ss, p, pp, op, os) =
  if(eq(p, s), m(pp, ss, op, os), n(os, op));

-- failover
-- current string is empty, so match fails
n("", op) = False();
-- trying the rest of the string
n(s:ss, op) = m(op, ss, op, ss);

-- equality routines
eq('A', y) = eqA(y);  eqA('A') = True();
eqB('A') = False();
eq('B', y) = eqB(y);  eqA('B') = False();
eqB('B') = True();
-- if/else
if(True(), x, y) = x;
if(False(), x, y) = y;
```

Figure 6: `prog2`: find a substring inside a string

analogous in action to the well-known Knuth-Morris-Pratt algorithm (hence the name of the test).

Our original simple transformer `transform` is unable to obtain an efficient matching program. But in the next couple of sections, we will augment `transform` using two "tricks", which are key ingredients of "real" supercompilers, and the transformer `supercompile` produces a residual task exactly corresponding to this DFA.

This program is the so-called KMP-test, which is a classical example in the context of supercompilation, as it demonstrates its greater power compared to

similar program transformations like deforestation and partial evaluation[6, 9]. This test is not only a good demonstration of the combined effect of the two tricks mentioned above, but it also gives some measure of the extent of program transformations performed by supercompilation.

Our original simple transformer `transform` gives the following result:

```
(match("AAB", s), prog2) ⟹(transform) (f1(s), prog2T)
```

where prog2T is:

```
f1(s) = f2(s);
f2(s) = g3(s, s);
g3("", s) = False();
g3(v1:v2, s) = f4(v1, v2, s);
f4(v1, v2, s) = g5(v1, v2, s);
g5('A', v2, s) = f6(v2, s);
g5('B', v2, s) = f22(v2, s);
f6(v2, s) = f7(v2, s);
f7(v2, s) = g8(v2, s);
g8("", s) = False();
g8(v3:v4, s) = f9(v3, v4, s);
f9(v3, v4, s) = g10(v3, v4, s);
g10('A', v4, s) = f11(v4, s);
g10('B', v4, s) = f20(v4, s);
f11(v4, s) = f12(v4, s);
f12(v4, s) = g13(v4, s);
g13("", s) = False();
g13(v5:v6, s) = f14(v5, v6, s);
f14(v5, v6, s) = g15(v5, v6, s);
g15('A', v6, s) = f16(v6, s);
g15('B', v6, s) = f18(v6, s);
f16(v6, s) = g17(s);
g17("") = False();
g17(v7:v8) = f2(v8);
f18(v6, s) = f19(v6, s);
f19(v6, s) = True();
f20(v4, s) = g21(s);
g21("") = False();
g21(v5:v6) = f2(v6);
f22(v2, s) = g23(s);
g23("") = False();
g23(v3:v4) = f2(v4);
```

This long listing is reproduced here only to serve as a baseline for comparison. Note that the residual program contains no nested function calls, but each of the

characters of `s` may be checked multiple times, as in the input program.

Transient step removal

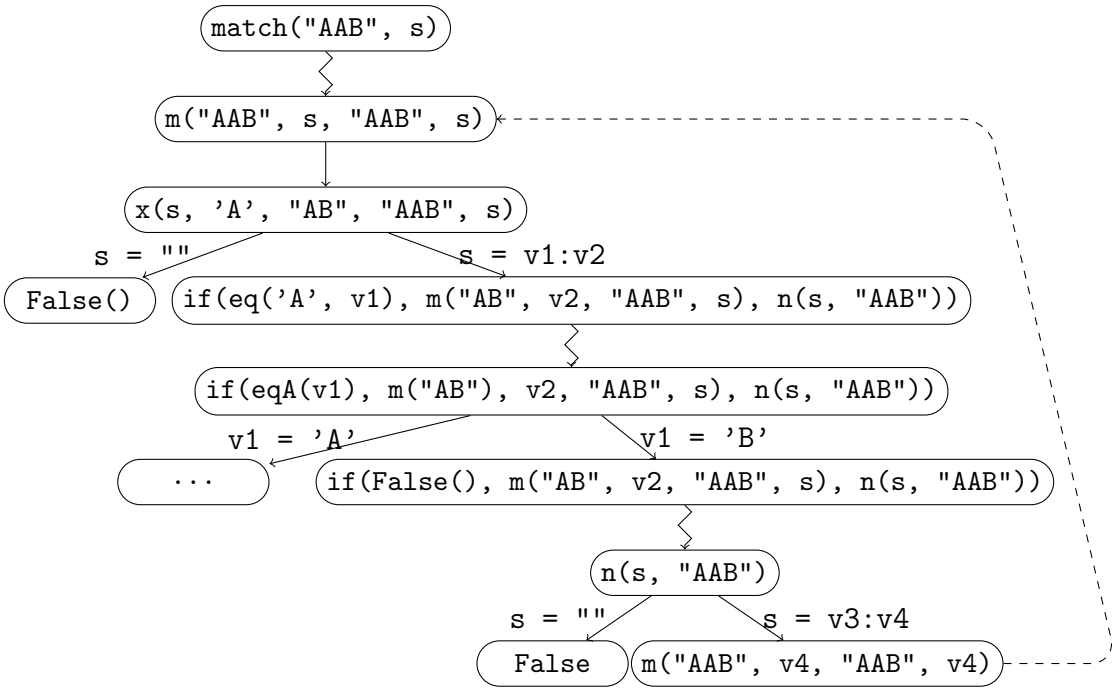
Very often graphs of configurations contain fragments of the following kind:

```
... c1 → c2 → c3 ...
```

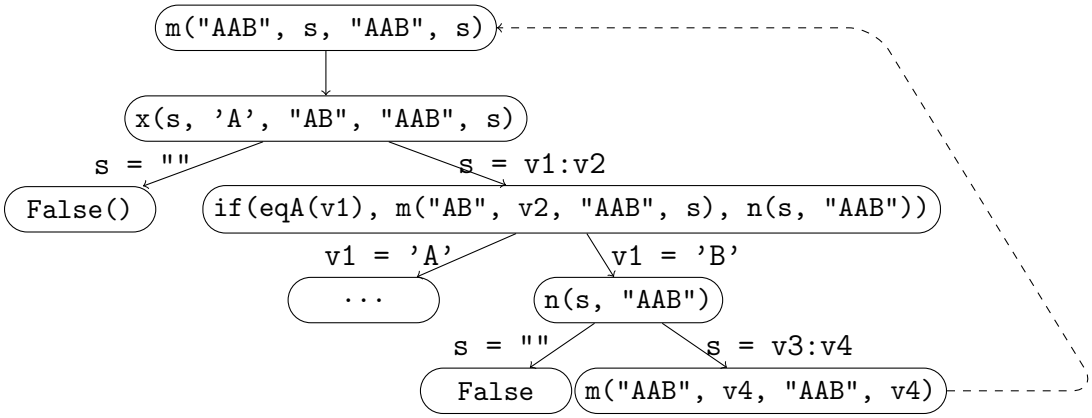
where the transition `c2 → c3` corresponds to a transient step (as defined in subsection "Driving"). We can always – thanks to the absence of side effects in SLL – replace such a fragment by:

```
... c1 → c3 ...
```

As an example, consider the top part of the graph of configurations for the KMP-test program, which is produced by `transform`:



The transient steps are displayed with zigzag arrows. If we remove them, the cleaned-up graph of configurations will look like this:



The SC Mini sources contain a transformer `deforest` – a modification of `transform` – which removes transient edges and the corresponding nodes from the graph of configurations.

If we pass the test program through `deforest`, we get:

`(match("AAB", s), prog2)` $\xRightarrow{deforest}$ `(f1(s), prog2D)`

where `prog2D =`

```
f1(s) = g2(s, s);
g2("", s) = False();
g2(v1:v2, s) = g3(v1, v2, s);
g3('A', v2, s) = g4(v2, s);
g3('B', v2, s) = g10(s);
g4("", s) = False();
g4(v3:v4, s) = g5(v3, v4, s);
g5('A', v4, s) = g6(v4, s);
g5('B', v4, s) = g9(s);
g6("", s) = False();
g6(v5:v6, s) = g7(v5, v6, s);
g7('A', v6, s) = g8(s);
g7('B', v6, s) = True();
g8("") = False();
g8(v7:v8) = f1(v8);
g9("") = False();
g9(v5:v6) = f1(v6);
g10("") = False();
g10(v3:v4) = f1(v4);
```

The deforested program contains less than half the number of functions (23 vs. 10) compared to the result of `transform`, as some intermediate functions were removed (roughly speaking, inlined). Although the residual program is smaller, it still contains the same inefficiency we discussed above.

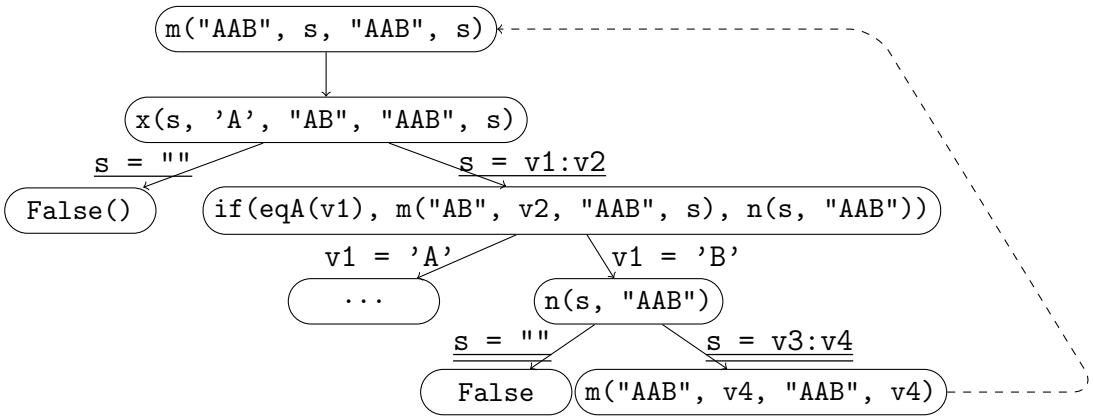
The removal of transient edges is a form of simplification of graphs of configurations. This simplification is **one of the two** main mechanisms for optimization, which are employed by supercompilation. It is most effective when used in conjunction with the second mechanism – information propagation – which we shall discuss next.

Exercise 13. It is also possible to remove transient steps from the tree of configurations (before folding it to a graph). What pitfalls might such an approach have?

There is a separate program transformation technique called **deforestation** [10, 11]. Its main goal is to reduce the creation of intermediate data structures – lists, trees (hence the name), etc. – which arise as a result of composing different functions in a program. Ferguson et al. [11] describe deforestation for a language very similar to SLL. Classical deforestation does not require generalization, as it considers only programs conforming to certain syntactic restrictions (which guarantee that the resulting tree is always foldable). A practical advantage of deforestation is that for many special cases it can be formulated as a simple set of rewrite rules (**shortcut deforestation**, or **shortcut fusion** – see for example [12]), which are easy to implement. A form of shortcut deforestation is implemented inside GHC, for example.

Information propagation

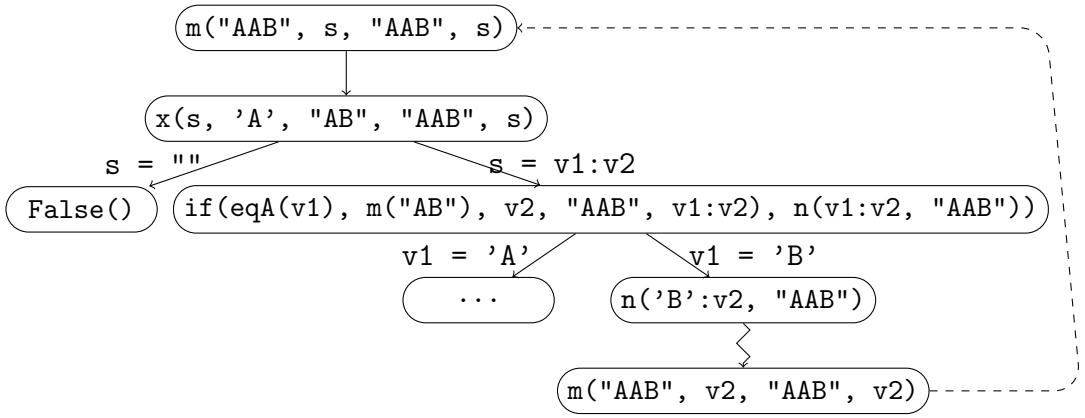
Here is the graph of configurations for the KMP test produced by `deforest`:



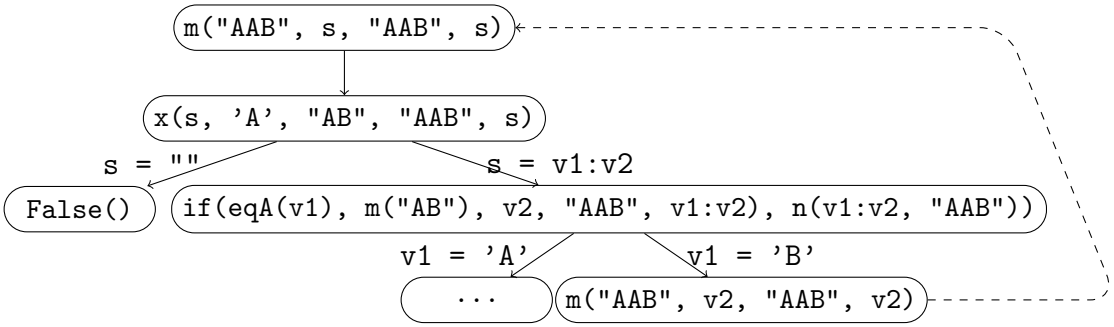
It contains a step which checks if the string `s` is empty or not (underlined). Below it, there is another step, which makes the same check (double-underlined), although we have assumed already that `s` is not empty (in the corresponding subtree after the first check). Such repeated checks are clearly redundant.

Supercompilers remove such redundancies as follows: each time case analysis is performed during driving, information about its outcome is propagated in each corresponding subtree.

Let's propagate the fact that `s = v1:v2` (and hence non-empty) in the corresponding subtree:



We have not only removed a superfluous test, but we have also unmasked a transient step, which can be further removed. As a result we obtain:



The transformer `supercompile` (inside SC Mini sources) implements such information propagation. In contrast to `deforest`, each time driving performs case analysis, its results are propagated in the corresponding new configurations. Recall that SLL contains just a single kind of test – matching the first argument of a curious function against a list of simple patterns. Hence, the only kind of information we can propagate is that a given configuration variable is equal to a given pattern. Such a (restricted) form of information propagation is called **positive information propagation**. Correspondingly, supercompilers propagating only positive informtion are called **positive supercompilers**. If, for example, SLL permitted also default clauses (in curious-function definitions), we could propagate also negative information – in the default clause – that the variable is not equal to any

of the patterns in the list. A supercompiler, which propagates both positive and negative information, is called **perfect** [13]. It is also possible to have intermediate situations, where only some forms of negative information are propagated.

Here is the final result of supercompiling our KMP test:

`(match("AAB", s), prog2)` $\xrightarrow{supercompile}$ `(f1(s), prog2S)`

where `prog2S =`

```
f1(s) = g2(s);
g2("") = False();
g2(v1:v2) = g3(v1, v2);
g3('A', v2) = g4(v2);
g3('B', v2) = f1(v2);
g4("") = False();
g4(v3:v4) = g5(v3, v4);
g5('A', v4) = f6(v4);
g5('B', v4) = f1(v4);
f6(v4) = g7(v4);
g7("") = False();
g7(v5:v6) = g8(v5, v6);
g8('A', v6) = f6(v6);
g8('B', v6) = True();
```

We finally obtain the desired effect: each character of the string `s` is considered at most once.

Exercise 14. Prove the last statement.

The information propagation we have just described is the second optimization mechanism employed by supercompilation (the first was the removal of transient steps). This propagation has 3 important effects:

1. No variable is tested twice.
2. Avoiding repeated tests uncovers further possibilities for removing transient steps.
3. The graph of configurations is pruned of unreachable branches, which results in dead-code removal in the residual program.

In the example above, the graph built by `deforest` contains the following path:

`{s = v1:v2}` $\rightarrow$ `...` $\rightarrow$ `{s = ""}` $\rightarrow$

It is clear that no actual computation can take this path. Deforestation, however, leaves it in the residual program; as a result the deforested program `prog2d` above contains a function `g10` whose first argument can never be an empty string:

```
g10("") = False();
g10(v3:v4) = f1(v4);
```

As a result of supercompiling the KMP test, we obtained a program which contains no dead code. Of course, this is not always possible, as dead-code removal is undecidable in general. In the case of supercompilation, dead code may still appear in the residual program if we have applied generalization.

Exercise 15. Most works discussing supercompilation interweave information propagation with other aspects of the transformation (for example, with transient-step removal). Check if this is the case as well inside SC Mini's sources.

The essence of SC Mini

This section turned out rather long, so let's take a brief look – from a slightly different angle – at what it covered.

The SC Mini supercompiler is an attempt to build a minimalistic supercompiler in Haskell. Its main goal is to delineate the main ingredients which are present in most supercompilers and show how they can be combined to work well together.

Let's review the gradual transition from the basic transformer `transform` (which neither improves, nor degrades performance), passing through `deforest`, and finally arriving at the ultimate transformer `supercompile`.
Starting from the baseline transformer `transform`...

```
transform :: Task -> Task
transform (e, p) =
    residuate $ foldTree $
        buildFTree (driveMachine p) e
```

...we add simplification of the graph of configurations by removing transient steps...

```
deforest :: Task -> Task
deforest (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (driveMachine p) e
```

...and we finish by adding information propagation:

```
supercompile :: Task -> Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (addPropagation  $ driveMachine p) e
```

Thanks to propagating test outcomes in the corresponding consequent configurations and further simplifications of the graph, our minimalistic supercompiler managed to convert a naïve substring search algorithm into the well-known efficient KMP algorithm.

Not only optimization

SC Mini can also be used to check different program properties. Let's show that the operation `add` – defined in `prog1` – is associative, that is, for all arguments `args`:

```
sll_run (add(add(x,y),z), p1) == sll_run (add(x,add(y,z), p1)
```

We could of course prove this by induction (using the rules of SLL operational semantics). But we can also compare the residual programs produced by SC Mini in both cases – they turn out syntactically equivalent:

```
(add(add(x, y), z), p1) ⟹supercompile (g1(x, y, z), prog3S')
(add(x, add(y, z)), p1) ⟹supercompile (g1(x, y, z), prog3S')
```

where `prog3S' =`

```
g1(Z(), y, z) = g2(y, z);
g1(S(v1), y, z) = S(g1(v1, y, z));
g2(Z(), z) = z;
g2(S(v1), z) = S(g2(v1, z));
```

This is enough to prove associativity of `add` (assuming of course that SC Mini always preserves the semantics of the input program), as we have:

```
sll_run (add(add(x, y), z), p1) ==
  sll_run (g1(x, y, z), prog3S') ==
    sll_run (add(x, add(y, z)), p1)
```

Problems

The aim of this section is to outline what are the practical problems our simple supercompiler faces. These problems are typical – in varying degrees – for most existing supercompilers.

Result unpredictability

The KMP test we saw indeed shows supercompilation at its best. We were lucky that the whistle did not blow, allowing us to fold the tree of configurations to a graph without resorting to generalization. In many other cases, however, the size of configurations in the tree will continue to grow, and, sooner or later, we shall be forced to perform generalization in order to ensure building a finite graph of configurations.

Recall a task we have already seen:

```
(even(sqr(x)), prog1)
```

Deforestation manages to build a finite graph of configurations without applying generalization. This is not the case when we use supercompilation instead – generalization becomes necessary. This is one drawback of information propagation, as it typically increases the size of the new configuration. The appendix accompanying this article lists the results of deforesting and of supercompiling this task. It also gives a comparison of the speed of the corresponding residual programs. Based on these observations, we can draw the following conclusions:

1. The supercompiled program is much larger than the deforested one.
2. For small numbers `x` the supercompiled program works faster. Exactly the opposite is true for big numbers `x` however.

In a way, this example is an antithesis of the KMP test, as it displays the main weaknesses of the SC Mini supercompiler. (We shall call it anti-KMP-test for brevity. Such an example can be found for each of the other existing supercompilers.)

While missing optimization opportunities is a feature we can readily consider acceptable, the fact that the supercompiler can produce much larger residual program, compared to the input one, is a more critical issue. The danger of code explosion is an important problem for supercompilation in general. In the case of SC Mini, by limiting configuration size we indirectly limit the growth of graph width as well. There is no similar indirect limit to the growth of graph depth (apart for ensuring it is finite), and this can result in very large residual programs.

Exercise 16. Try to modify SC Mini, by imposing an explicit limit on graph depth as well. What changes in the examples we considered so far?

Exercise 17. Can we estimate the size of the residual program for a given input?

SC Mini is built in such a way as to guarantee that the residual program never takes more reduction steps than the input one. This is but one possible measure of efficiency. Another one is memory consumption, and here the story is not so rosy. Existing supercompilers can produce programs, which – although requiring less reduction steps – can sometimes consume considerably more memory than the original ones.

Scalability issues

The supercompiler imitates the behavior of the input program by taking into account the dynamic interactions of all its parts. This can result, unfortunately, in the size of the model (the graph of configurations), growing very superlinearly as a function of the size of the input program. For example, a new input program, larger by just 30%, can make the supercompiler work 10 times longer.

Exercise 18. Try to find such examples for SC Mini.

What is worse, the running time of the supercompiler usually depends superlinearly on the size of the graph of configurations.

Exercise 19. Show that this statement is true for SC Mini as well.

Supercompiling big programs can be a very unpredictable process: in the worst case, the supercompiler may take a lot of time, and it may produce a huge program, which works only slightly faster than the original.

Supercompilation performs a global optimization for a given entry point. It is an open question how to apply supercompilation methods to optimize libraries.

What about debugging?

Supercompilation – being a form of program transformation – converts the input program into a new one, which is then submitted to the usual programming language compiler or interpreter. What if we want to debug the resulting program? It appears possible in principle – in a way similar to standard compilers generating "debug" info – to generate some information about the correspondence of residual program lines to input program lines. This can be a tedious task, however, and authors of experimental supercompilers avoid it. We can hope that one day industrial-strength supercompilers will support this feature as well.

Details, details, details...

We saw only a very small, toy supercompiler – for a toy language – which has some obvious deficiencies. Building a supercompiler for a similar toy language, but not having the problems we mentioned, is already a complicated, PhD-level task. If we consider more realistic programming languages, we will quickly stumble upon a number of details, which must be taken care of.

Global state, side effects: SC Mini uses the compositionality of SLL semantics in an essential way in order to perform generalization:

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

The expression e_2 may be evaluated outside of the context, where it appears in the input program.

If the language features global state and/or side effects (which is inevitable in one form or another for any practical programming language), then the notion of compositionality becomes more complicated, if it can be formulated at all. In order to take this into account, an additional analysis phase appears necessary. Things

can get even more complicated in the presence of concurrency. What makes the situation less bleak is that many modern programming languages follow the lead of Haskell and provide means to isolate pure functional code from side-effecting code. In such cases a simple solution is to supercompile only the purely functional parts of the program, leaving the side-effecting parts unmodified.

Taking strictness and laziness into account: SC Mini owes its simplicity, to a large extent, to the fact that we assumed a call-by-name semantics for SLL. Driving is simplest in the case of call-by-name evaluation. Call-by-name typically leads to bad performance, however, and most programming languages do not use it as the default evaluation mechanism. Call-by-value or call-by-need is used instead. While recent research has shown it is possible to build a supercompiler for a call-by-need [14, 15, 16] or a call-by-value [17, 18] language, the supercompilation process is typically much more involved, compared to the case of call-by-name.

Exercise 20. Write a call-by-value interpreter for SLL. Find a program, which behaves differently after supercompilation by SC Mini, when run by the new interpreter.

Exercise 21. Write a call-by-need interpreter for SLL. Compare its size to the call-by-name one. What differences in driving would you expect in the case of call-by-need?

Why is it worth it then?

The open problems of supercompilation are wide, the results of supercompilation difficult to rely on. Why is it, then, that interest in supercompilation has repeatedly resurfaced and grown in the several decades since its invention, with the latest wave of renewed interest starting 6-7 years ago [19, 20, 16, 18, 21, 22, 23]?

First of all, it turns out that many program optimization methods can be seen as special cases of supercompilation. This includes deforestation, partial evaluation, different kinds of fusion, inlining, defunctionalization, etc. (While this statement will be intuitively obvious to most researchers in the area of supercompilation, not all of these cases have been formally described in research papers.) It is most impressing when a (relatively) simple supercompiler can optimize some programs equally well (or better!) than some other complicated specialized tool.

A second important reason is the conceptual simplicity of supercompilation. Supercompilation is not tied to a single specific language, nor even to a family of languages, although most of the existing research has been done in the context of functional languages.

Another valuable feature of supercompilation is that it has applications beyond optimization. There are very successful attempts to apply it to program analysis,

for example. It is tempting to image a 2-in-1, or even 3-in-1 tool, which can cover program optimization, analysis, synthesis, etc.

Last but not least, supercompilation is itself just a specialized application of a very general philosophical principle, invented also by V. F. Turchin – the theory of "metasystem transitions". We shall speak a bit more about that in the next section.

A little history

Valentin Fyodorovich Turchin (1931–2010) – the creator of supercompilation – could be called a Programmer-Philosopher.

"Valentin Fedorovich Turchin, born in 1931, holds a doctor's degree in the physical and mathematical sciences. He worked in the Soviet science center in Obninsk, near Moscow, in the Physics and Energetics Institute and then later became a senior scientific researcher in the Institute of Applied Mathematics of the Academy of Sciences of the USSR. In this institute he specialized in information theory and the computer sciences. While working in these fields he developed a new computer language that was widely applied in the USSR, the "Refal" system. After 1973 he was the director of a laboratory in the Central Scientific-Research Institute for the Design of Automated Construction Systems. During his years of professional employment Dr. Turchin published over 65 works in his field. In sum, in the 1960s and early 1970s, Valentin Turchin was considered one of the leading computer specialists in the Soviet Union." (from L. R. Graham's foreword to "The phenomenon of science" by Turchin [24].)

"The intellectual pivot of the book is the concept of the metasystem transition – the transition from a cybernetic system to a metasystem, which includes a set of systems of the initial type organized and controlled in a definite manner. I first made this concept the basis of an analysis of the development of sign systems used by science. Then, however, it turned out that investigating the entire process of life's evolution on earth from this point of view permits the construction of a coherent picture governed by uniform laws. . . ." (from the introduction to "The phenomenon of science" by Turchin [24].)

In 1966, Turchin invented Refal (REcursive Functions Algorithmic Language) – a programming language that was quite different from most other existing ones. Refal was oriented towards describing and processing other languages. Even a brief description of Refal is beyond the scope of this article, let's just simply mention that it quickly gathered a small group of active supporters which met regularly in Moscow. The next 5–6 years were devoted to creating an efficient implementation of Refal – first an interpreter, and later a compiler as well. The compiler was itself written in Refal: it took Refal programs as input and generated assembly

programs as output. But nothing prevented the generation of Refal as output as well. This was how the idea of driving was born, which Turchin originally described as "equivalent transformations of Refal functions". In 1974, driving was described already in the context of the theory of metasystem transitions.

In 1977, Turchin was forced to leave the Soviet Union; in the same year an English translation of Turchin's "opus magnum" – the book "The phenomenon of science" – was published in the USA. This book was already finished in 1970 and ready for printing in 1973, but its publication in the Soviet Union was blocked for political reasons. Since then, Turchin lived and worked in New York, first in the Courant Institute, later in the City College. Starting from 1989, Turchin was again able to visit Russia, which he did regularly till his death in 2010.

We can distinguish – quite subjectively – 3 periods in the history of supercompilation.

- **1970s–1980s. Refal supercompilers.** Starting from 1979, Turchin published (with coauthors) several tens of articles on supercompilation and metacomputation. It is now obvious that a huge number of interesting ideas lies scattered inside these articles. The problem was that many concepts were given in only a semi-formal, fragmentary way, and only in terms of Refal: for Turchin, Refal and supercompilation were inseparable. Unfortunately, for most people at the time even the description of driving seemed too complicated and incomprehensible. And no article contained a full and self-contained description of the complete process of supercompilation. For these reasons, in spite of the large number of publications on the subject, till the early 1990s supercompilation remained understood and appreciated only by a small number of "initiates". Turchin's articles [5, 25, 3, 26, 27, 28, 29] are some of the milestones of this period.

- **1990s. Supercompilation of first-order functional languages.** Andrei Klimov's and Robert Glück's article "Occam's Razor in Metacompuation: the Notion of a Perfect Process Tree" [30] about the essence of driving was the first work aimed at understanding supercompilation as a general technique, independent of Refal. In 1994 Morten H. Sørensen made an important further step – in his MSc thesis, "Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation," [6] he reformulated the key ideas of supercompilation in the context of a simple first-order functional language (essentially the same as SLL). This was the first work describing the process of supercompilation in full. It was followed by a number of other articles [31, 32, 33, 34, 9, 35] explaining supercompilation and comparing it to other program-transformation methods.

- **2000s. Supercompilation of higher-order functional languages.** The latest wave of renewed interest in supercompilation started in the second half of the 2000s. Many new milestones were passed (supercompilers for call-by-need, call-by-value, etc.), but the single most important shift so far has been from first-order to higher-order functional languages. Regular international workshops on supercompilation are being held – META-2008 [36], META-2010 [37], META-2012 [38], META-2014 [39].

Current trends

This is only an introductory article on supercompilation, and a detailed survey of the current state of the field is beyond its scope. Instead, we list some existing supercompilers in the next section. Very briefly, one of the main goals in current research is to build a supercompiler which can transcend the experimental status and become practically useful for a larger audience. Another important trend is the application of the ideas of supercompilation in new contexts.

If you are curious to see a more detailed picture of the current state of supercompilation research, many recent PhD theses and other articles [6, 40, 41, 14, 42, 17, 43, 18, 44] contain very good overview sections.

Existing supercompilers

We list here some existing supercompilers. The list is not exhaustive; it contains implementations which have interesting features, are publicly accessible, and which are either actively developed, or at least have been until recently. Note that all these implementations are considered experimental.

- **SCP4 [45].** SCP4 is the latest incarnation of the first ever supercompiler, which was developed for the Refal language. It supercompiles programs in a recent version of Refal, Refal-5. SCP4 utilizes some of the features of Refal, which make it particularly suited to supercompilation, like the associativity of sequence concatenation. It also features a number of extensions of the basic super-compilation method: recognition of constant functions, recognition of concatenation monomials, and collection and analysis of output formats. SCP4 is described in detail in [42] and in the monograph [46]. SCP4 can extend the program domain in the following sense: if the input program loops or stops with an error on certain inputs, it is sometimes possible for the residual program to successfully terminate on these inputs.

- **A (nameless) supercompiler for the TSG language [47].** TSG is a greatly simplified version of LISP, which is "flat" (no nested function calls are al-

lowed). Besides the experimental supercompiler for TSG described in [35], there exist implementations for a number of other methods, which have arisen in the context of supercompilation, and which, unfortunately, get less attention than they deserve. These methods include neighborhood analysis, neighborhood testing, inverse programming, and nonstandard semantics [31].

- **Jscp[48] [49].** A supercompiler for Java and the first supercompiler for an object-oriented, real-world language. One of the main conclusions of this experiment: supercompilation of non-functional languages is much more complicated. Jscp, unlike most other supercompilers in this list, is closed-source.

- **A supercompiler for Timber [50].** Timber is a pure object-oriented language with call-by-value semantics mostly inspired by Haskell. Its supercompiler treats the purely functional subset of the language. The main goal of this project is to achieve similar optimizations when supercompiling a call-by-value language – compared to a call-by-name language – while fully preserving the semantics of the original program. Good descriptions exist in [17, 18].

- **Supero [51].** A supercompiler for a subset of Haskell. Supero is the first supercompiler to treat a call-by-need language, which was actually the main goal of the project [14, 15].

- **SPSC [52].** SPSC is another toy supercompiler for SLL. The main goal was to implement positive supercompilation, as described (but not implemented) by Sørensen [34, 9]. SPSC is described in [53].

- **HOSC [54].** HOSC is a supercompiler for a (call-by-name) Haskell subset. Unlike many other projects, where the main goal is program optimization, here the main interest lies in program analysis by supercompilation. Besides standard supercompilation, HOSC can also perform "two-level" supercompilation, based on discovering and applying "improvement" lemmas [43, 55].

- **Optimusprime [56].** Optimizes functional programs, which are then run on a specialized FPGA-build processor (Reduceron) [22].

- **CHSC [57].** Another supercompiler for a Haskell subset [16]. The main goal of this project was to include supercompilation as an optimization pass inside GHC. An important technical difference is that CHSC does not perform lambda-lifting as a preprocessing step, unlike most other supercompilers for higher-order languages.

- **Distillation [58].** Most supercompilers use configurations based on expressions with free variables. Distillation uses configurations, each of which in

turn is similar to a configuration graph. In other words, the nodes of distillation's configuration graphs contain nested graphs; and folding and generalization must be defined over graphs. Such complications are the price for obtaining more powerful optimizations, compared to standard supercompilation [59, 60].

- **MRSC [61]** A toolkit for building "multi-result" supercompilers. Multi-result supercompilation [62] is another generalization of the standard supercompilation technique, where the supercompiler is permitted to return multiple residual programs (all correct, of course). While this may seem useless at first glance, it turns our that this extension opens the way for a much more flexible and modular description of supercompilation, and in the same time permits some optimizations, which are out of reach for the standard methods.

- **A Coq framework for building formally verified supercompilers [63]** A framework in Coq for building modular supercompilers by just defining the basic ingredients (configurations, driving, folding, etc.), and plugging them into a prefabricated generic supercompiler [64]. The organization of the supercompiler is very similar to the one we have just described, and there is a complete example supercompiler for a language very close to SLL. The main advantage of the framework is that it facilitates formally establishing the correctness of each new supercompiler, by just having to prove some simple properties concerning the basic building blocks.

- **A supercompiler for Erlang [65]** A recent first step towards a practical supercompiler for another popular language, Erlang.

- **TT-Lite [66]** A supercompiler for a version of Martin-Löf's Type Theory (MLTT). TT-Lite is the first supercompiler for a terminating (non-Turing-complete) language with dependent types [67]. Another interesting feature is the generation of "certificates", containing formal, automatically verifiable evidence that the residual program is equivalent to the input one in each specific instance. The certificates themselves are encoded as MLTT terms.

Instead of a conclusion

The main driving force leading to this article was the following idea: using Haskell to describe – in a clear and modular way – the main ingredients of a minimalistic supercompiler and show how they fit together. As a result we arrived at the following definition:

```
supercompile :: Task -> Task
```

```
supercompile (e, p) =
        residuate $ simplify $ foldTree $
                buildFTree (addPropagation $ driveMachine p) e
```

The text of the article is, in fact, just an attempt to describe each of the parts of this definition and to show what effects they can achieve together.

We hope the reader will take some time to study SC Mini's sources, where comments describe some interesting technical details of the implementation.

Where can we go from here?

If you are interested in learning more about supercompilation, here is a short list of possible next steps:

1. The articles [9, 34] are still very good and readable introductions to the standard techniques used in supercompilation, including some not covered here, such as using homeomorphic embedding as a whistle and using "most specific generalization" as a generalization algorithm.
2. the Google group "Supercompilation and Related Techniques" [68] contains the most up-to-date discussions and announcements concerning supercompilation.

References

[1] Ilya Klyuchnikov and Dimitur Krustev. The Sc Mini Supercompiler. **The Monad Reader**, (23) (2014). (Appendix to the article "Supercompilation: Ideas and Methods").

[2] `https://github.com/ilya-klyuchnikov/sc-mini`.

[3] V. F. Turchin. The concept of a supercompiler. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, 8(3):pages 292–325 (1986).

[4] Simon L. Peyton Jones. Interview. **Practice of functional programming**, (6) (2010). (In Russian) `http://fprog.ru/2010/issue6/interview-simon-peyton-jones/`.

[5] Turchin V.F. Equivalent transformations of refal programs. Trudy CNIPIASS 6, CNIPIASS (1974).

[6] M. H. Sørensen. **Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation**. Master's thesis, Københavns Universitet, Datalogisk Institut (1994).

[7] V. F. Turchin. Program transformation by supercompilation. In **Programs as Data Objects**, volume 217 of **LNCS**, pages 257–281. Springer (1986).

[8] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. **SIAM Journal on Computing**, 6:page 323 (1977).

[9] M. H. Sørensen and R. Glück. Introduction to supercompilation. In **Partial Evaluation. Practice and Theory**, volume 1706 of **LNCS**, pages 246–270 (1998).

[10] P. Wadler. Deforestation: Transforming programs to eliminate trees. In **ESOP '88**, volume 300 of **LNCS**, pages 344–358. Springer (1988).

[11] A.B. Ferguson and P. Wadler. When Will Deforestation Stop? In **1988 Glasgow Workshop on Functional Programming** (1988).

[12] A. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. **Proceedings of the conference on Functional programming languages and computer architecture**, pages 223–232 (1993).

[13] J. P. Secher and M. H. Sørensen. On perfect supercompilation. In **PSI '99**, volume 1755 of **LNCS**, pages 113–127. Springer-Verlag London, UK (2000).

[14] N. Mitchell. **Transformation and Analysis of Functional Programs**. Ph.D. thesis, University of York (2008).

[15] N. Mitchell. Rethinking supercompilation. In **ICFP 2010** (2010).

[16] Max Bolingbroke and Simon L. Peyton Jones. Supercompilation by evaluation. In **Haskell 2010 Symposium** (2010).

[17] P.A. Jonsson. **Positive supercompilation for a higher-order call-by-value language**. Licentiate thesis, Luleå University of Technology (2008).

[18] P.A. Jonsson. **Positive Supercompilation for Higher-Order Languages**. Ph.D. thesis, Luleå University of Technology (2011).

[19] Alexei Lisitsa and Andrei Nemytykh. Towards verification via supercompilation. In **COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference**, pages 9–10. IEEE Computer Society, Washington, DC, USA (2005).

[20] N. Mitchell and C. Runciman. A supercompiler for core haskell. In **Implementation and Application of Functional Languages**, volume 5083 of **Lecture Notes In Computer Science**, pages 147–164. Springer-Verlag, Berlin, Heidelberg (2008).

[21] Ilya Klyuchnikov and Sergei Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In **Perspectives of Systems Informatics**, volume 5947 of **LNCS**, pages 193–205 (2010).

[22] Jason S. Reich, Matthew Naylor, and Colin Runciman. Supercompilation and the Reduceron. In **Proceedings of the Second International Workshop on Metacomputation in Russia** (2010).

[23] Max Bolingbroke and Simon L. Peyton Jones. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation. In **ICFP 2011** (2011).

[24] V. F. Turchin. **The phenomenon of science. A cybernetic approach to human evolution**. Columbia University Press, New York (1977). `http://pespmc1.vub.ac.be/POSBOOK.html`.

[25] V. F. Turchin. **The Language Refal: The Theory of Compilation and Metasystem Analysis**. Department of Computer Science, Courant Institute of Mathematical Sciences, New York University (1980).

[26] V. F. Turchin. The algorithm of generalization in the supercompiler. In **Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop** (1988).

[27] V. F. Turchin. Program transformation with metasystem transitions. **Journal of Functional Programming**, 3(03):pages 283–313 (1993).

[28] V. F. Turchin. Supercompilation: Techniques and results. In **Perspectives of System Informatics**, volume 1181 of **LNCS**. Springer (1996).

[29] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In **Partial Evaluation**, volume 1110 of **Lecture Notes in Computer Science**, pages 481–509. Springer (1996).

[30] Robert Glück and A.V. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In **WSA '93: Proceedings of the Third International Workshop on Static Analysis**, pages 112–123. Springer-Verlag, London, UK (1993).

[31] Abramov, S. M. **Metavychisleniya i ih primenenie (Metacomputation and its applications)**. Nauka (1995). In Russian.

[32] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd (editor), **Logic Programming: The 1995 International Symposium**, pages 465–479 (1995).

[33] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In **Selected Papers From the International Seminar on Partial Evaluation**, volume 1110 of **LNCS**, pages 137–160 (1996).

[34] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. **Journal of Functional Programming**, 6(6):pages 811–838 (1996).

[35] Abramov, S. M. and Parmyonova, L. V. **Metavychisleniya i ih primenenie. Superkompiliacia (Metacomputation and its applications. Supercompilation, in Russian)**. Program Systems Institute of the RAS (2006). In Russian.

[36] Program Systems Institute. **First International Workshop on Metacomputation in Russia** (July 2–5 2008). http://meta2008.pereslavl.ru/.

[37] Program Systems Institute. **Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia** (July 1-5 2010). http://meta2010.pereslavl.ru/.

[38] Program Systems Institute. **Third International Valentin Turchin Workshop on Metacomputation** (July 5-9 2012). http://meta2012.pereslavl.ru/.

[39] Program Systems Institute. **Fourth International Valentin Turchin Workshop on Metacomputation** (2014). http://meta2014.pereslavl.ru/.

[40] Jens Peter Secher. **Perfect Supercompilation**. Master's thesis, Department of Computer Science, University of Copenhagen (1999).

[41] J.P. Secher. **Driving-Based program transformation in theory and practice**. Ph.D. thesis, Department of Computer Science, Copenhagen University (2002).

[42] Nemytykh A.P. **Specialization of functional programs using supercompilation**. Phd thesis, Program Systems Institute of the RAS (2008). In Russian.

[43] Klyuchnikov I.G. **Inferring and proving properties of functional programs by means of supercompilation**. Phd thesis, M.V.Keldysh Institute of Applied Mathematics of the RAS (2010). In Russian, http://pat.keldysh.ru/~ilya/klyuchnikov-phd.pdf.

[44] Max Bolingbroke. **Call-by-need supercompilation**. Ph.D. thesis, Computer Laboratory, University of Cambridge (2013). http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-835.html.

[45] http://www.botik.ru/pub/local/scp/refal5/refal5.html.

[46] Nemytykh A.P. **Supercompiler SCP4. General structure**. LKI, Moscow (2007).

[47] http://www.botik.ru/~abram/mca/.

[48] A.V. Klimov. An approach to supercompilation for object-oriented languages: the java supercompiler case study. In **First International Workshop on Metacomputation in Russia** (2008).

[49] http://supercompilers.ru.

[50] http://timber-lang.org/.

[51] http://community.haskell.org/~ndm/supero/.

[52] http://code.google.com/p/spsc/.

[53] Ilya Klyuchnikov and Sergei Romanenko. SPSC: a Simple Supercompiler in Scala. In **PU'09 (International Workshop on Program Understanding)** (2009). http://spsc.googlecode.com/files/Klyuchnikov__Romanenko__SPSC_a_Simple_Supercompiler_in_Scala.pdf.

[54] http://code.google.com/p/hosc/.

[55] Ilya Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics (2010). http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e.

[56] http://hackage.haskell.org/package/optimusprime.

[57] https://github.com/batterseapower/chsc.

[58] https://github.com/distillation/distill.

[59] G. W. Hamilton. Distillation: extracting the essence of programs. In **Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation**, pages 61–70. ACM Press New York, NY, USA (2007).

[60] G. W. Hamilton. A graph-based definition of distillation. In **Second International Workshop on Metacomputation in Russia** (2010).

[61] https://github.com/ilya-klyuchnikov/mrsc.

[62] I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute (2011). http://library.keldysh.ru/preprint.asp?id=2011-77&lg=e.

[63] https://sites.google.com/site/dkrustev/Home/publications.

[64] Dimitur Krustev. Towards a framework for building formally verified supercompilers in Coq. In Hans-Wolfgang Loidl and Ricardo PeÃśa (editors), **Trends in Functional Programming**, volume 7829 of **Lecture Notes in Computer Science**, pages 133–148. Springer Berlin Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-40447-4_9.

[65] https://weinholt.se/.

[66] https://github.com/ilya-klyuchnikov/ttlite.

[67] Ilya Klyuchnikov and Sergei Romanenko. TT Lite: a supercompiler for Martin-Löf's type theory. Preprint, KIAM, Moscow (2013). http://library.keldysh.ru//preprint.asp?lg=e&id=2013-73.

[68] http://groups.google.com/forum/#!forum/supercompilation-and-related-techniques.

The SC Mini Supercompiler

by Ilya Klyuchnikov ⟨ilya.klyuchnikov@gmail.com⟩
and Dimitur Krustev ⟨dkrustev@gmail.com⟩

The internals of the supercompiler SC Mini are explained in this appendix. The actual code is at ***https://github.com/ilya-klyuchnikov/sc-mini*** *.*

Introduction

The SC Mini sources can – somewhat subjectively – be divided into the following parts:

1. Data type definitions with some basic operations: SLL abstract syntax; basic operations on SLL expressions; definition of a SLL "task" and "graph of configurations".
   - `Data.hs`
2. Auxiliary functions for working with the main data structures: parser, substitutions, expression comparison, etc.
   - `DataUtil.hs`
3. Main part – interpreter, driving, folding, transformers `transform`, `deforest` and `supercompile`, residual task generator.
   - `Interpreter.hs`
   - `Driving.hs`
   - `TreeInterpreter.hs`
   - `Folding.hs`
   - `Generator.hs`
   - `Prototype.hs`
   - `Deforester.hs`
   - `Supercompiler.hs`
4. A set of examples together with their results.
   - `Demonstration.hs`

The full SC Mini sources are listed, only the parsers/pretty-printers are omitted. New definitions are illustrated with actual output from running them. All interesting examples are collected in the module `Demonstration.hs`. The reader can also download and directly play with the actual sources.

For convenience, we recall here the definition of SLL operational semantics.

$$
\begin{array}{lcl}
con & ::= & \langle\rangle \mid g(con, \ldots) \\
red & ::= & f(e_1, \ldots, e_n) \mid g(C(e_1, \ldots, e_n), \ldots) \mid g(v, \ldots)
\end{array}
$$

Figure 1: SLL: expression decomposition

$$
\begin{array}{lcl}
\mathcal{I}[\![e]\!] & \Rightarrow & e \\
 & & \text{if } e \text{ is a value} \\
\mathcal{I}[\![C(e_1, \ldots, e_n)]\!] & \Rightarrow & C(\mathcal{I}[\![e_1]\!], \ldots, \mathcal{I}[\![e_n]\!]) \\
\mathcal{I}[\![con\langle f(e_1, \ldots, e_n)\rangle]\!] & \Rightarrow & \mathcal{I}[\![con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]\!] \\
 & & \text{if } f(x_1, \ldots, x_n) \stackrel{\text{p}}{=} e \\
\mathcal{I}[\![con\langle g(C(e_1, \ldots, e_m), e_{m+1}, \ldots, e_n)\rangle]\!] & \Rightarrow & \mathcal{I}[\![con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]\!] \\
 & & \text{if } g(C(v_1, \ldots, v_m), v_{m+1}, \ldots, v_n) \stackrel{\text{p}}{=} e
\end{array}
$$

Figure 2: Reduction step rules

Data.hs

First a definition of SLL abstract syntax. To make generalization easier, it already includes `let`-expressions.

```
type Name = String
data Expr = Var Name | Ctr Name [Expr] | FCall Name [Expr]
    | GCall Name [Expr] | Let (Name, Expr) Expr
    deriving (Eq)
data Pat = Pat Name [Name] deriving (Eq)
data GDef = GDef Name Pat [Name] Expr deriving (Eq)
data FDef = FDef Name [Name] Expr deriving (Eq)
data Program = Program [FDef] [GDef] deriving (Eq)
```

Type synonyms for operations on expressions: renaming, substitution, fresh-name supply.

```
type Renaming = [(Name, Name)]
type Subst = [(Name, Expr)]
```

```
11 type NameSupply = [Name]
```

Technically we work with SLL-expressions almost everywhere. But to distinguish in which sense we use them – expression, configuration, value – we introduce extra type synonyms:

```
12 type Conf = Expr
13 type Value = Expr
14 type Task = (Conf, Program)
15 type Env = [(Name, Value)]
```

Probably the most interesting data types:

```
16 data Contract = Contract Name Pat
17 data Step a = Transient a | Variants [(Contract, a)] | Stop
18     | Decompose [a] | Fold a Renaming
19 data Graph a = Node a (Step (Graph a))
20 type Tree a = Graph a
21 type Node a = Tree a
22
23 type Machine a = NameSupply -> a -> Step a
```

In the following we shall consider a program as a kind of machine. `Machine a` operates with some generalized states of type `a`. We assume further, that states can contain named subparts (using identifiers), and that the machine may need to produce some new named subparts. If the machine is given some infinite list of names and some current state, then it computes in one step a next generalized state. The data type `Step a` describes the kinds of steps we use – transient, final, decomposition, etc. The most interesting kind of step – `Variants [(c1, a1), (c2, a2),...]` – describes case analysis. Its meaning is: if the condition `c1` holds, then the next state will be `a1`, if `c2` holds – `a2`, etc. We consider only simple conditions of type `Contract` – namely that some variable matches a certain pattern. `Graph a` – is the graph of state transitions.

We shall use further only configurations as states, but the type of state is deliberately abstracted here for full generality.

DataUtil.hs

The module `DataUtil` defines some (mostly boring) auxiliary functions. Its text is included for completeness anyway.

Working with abstract syntax:

```
1 isValue :: Expr -> Bool
2 isValue (Ctr _ args) = and $ map isValue args
```

```
isValue _ = False

isCall :: Expr -> Bool
isCall (FCall _ _) = True
isCall (GCall _ _) = True
isCall _ = False

isVar :: Expr -> Bool
isVar (Var _) = True
isVar _ = False

fDef :: Program -> Name -> FDef
fDef (Program fs _) fname =
    head [f | f@(FDef x _ _) <- fs, x == fname]

gDefs :: Program -> Name -> [GDef]
gDefs (Program _ gs) gname =
    [g | g@(GDef x _ _ _) <- gs, x == gname]

gDef :: Program -> Name -> Name -> GDef
gDef p gname cname =
    head [g | g@(GDef _ (Pat c _) _ _) <-
        gDefs p gname, c == cname]
```

Applying substitutions:

```
(//) :: Expr -> Subst -> Expr
(Var x) // sub = maybe (Var x) id (lookup x sub)
(Ctr name args) // sub = Ctr name (map (// sub) args)
(FCall name args) // sub = FCall name (map (// sub) args)
(GCall name args) // sub = GCall name (map (// sub) args)
(Let (x, e1) e2) // sub  = Let (x, (e1 // sub)) (e2 // sub)
```

Working with names. `nameSupply` produces an infinite list of names. `unused` removes from the list of names those used inside a condition. `vnames'` – all the names inside an expression, `vnames` – the same without repetitions. `isRepeated` checks if a name occurs more than once in an expression.

```
nameSupply :: NameSupply
nameSupply = ["v" ++ (show i) | i <- [1 ..] ]

unused :: Contract -> NameSupply -> NameSupply
unused (Contract _ (Pat _ vs)) = (\\ vs)

vnames :: Expr -> [Name]
```

```
vnames = nub . vnames'

vnames' :: Expr -> [Name]
vnames' (Var v) = [v]
vnames' (Ctr _ args)   = concat $ map vnames' args
vnames' (FCall _ args) = concat $ map vnames' args
vnames' (GCall _ args) = concat $ map vnames' args
vnames' (Let (_, e1) e2) = vnames' e1 ++ vnames' e2

isRepeated :: Name -> Expr -> Bool
isRepeated vn e = (length $ filter (== vn) (vnames' e)) > 1
```

`renaming e1 e2` finds a renaming (if possible) from an expression `e1` into an expression `e2` (probably a more elegant definition is possible):

```
renaming :: Expr -> Expr -> Maybe Renaming
renaming e1 e2 =
    f $ partition isNothing $ renaming' (e1, e2) where
    f (x:_, _) = Nothing
    f (_, ps) = g gs1 gs2
        where
            gs1 = groupBy (\(a, b) (c, d) -> a == c)
                $ sortBy h $ nub $ catMaybes ps
            gs2 = groupBy (\(a, b) (c, d) -> b == d)
                $ sortBy h $ nub $ catMaybes ps
            h (a, b) (c, d) = compare a c
    g xs ys =
        if all ((== 1) . length) xs
            && all ((== 1) . length) ys
        then Just (concat xs) else Nothing

renaming' :: (Expr, Expr) -> [Maybe (Name, Name)]
renaming' ((Var x), (Var y)) =
    [Just (x, y)]
renaming' ((Ctr n1 args1), (Ctr n2 args2)) | n1 == n2 =
    concat $ map renaming' $ zip args1 args2
renaming' ((FCall n1 args1), (FCall n2 args2)) | n1 == n2 =
    concat $ map renaming' $ zip args1 args2
renaming' ((GCall n1 args1), (GCall n2 args2)) | n1 == n2 =
    concat $ map renaming' $ zip args1 args2
renaming' (Let (v, e1) e2, Let (v', e1') e2') =
    renaming' (e1, e1') ++
        renaming' (e2, e2' // [(v, Var v')])
renaming' _  = [Nothing]
```

Expression size (used inside the whistle):

```
size :: Expr -> Integer
size (Var _)          = 1
size (Ctr _ args)     = 1 + sum (map size args)
size (FCall _ args)   = 1 + sum (map size args)
size (GCall _ args)   = 1 + sum (map size args)
size (Let (_, e1) e2) = 1 + (size e1) + (size e2)
```

Other utility functions:

```
nodeLabel :: Node a -> a
nodeLabel (Node l _) = l

step :: Node a -> Step (Graph a)
step (Node _ s) = s
```

Interpreter.hs

The module `Interpreter` defines an evaluator for SLL expressions.

The interpreter `int` works in "small-step" fashion, or in other words, reduction steps are repeated until the expression becomes a value. `intStep` implements a single reduction step; note that it is not recursive.

```
int :: Program -> Expr -> Expr
int p e = until isValue (intStep p) e

intStep :: Program -> Expr -> Expr
intStep p (Ctr name args) =
    Ctr name (values ++ (intStep p x : xs)) where
        (values, x : xs) = span isValue args

intStep p (FCall name args) =
    body // zip vs args where
        (FDef _ vs body) = fDef p name

intStep p (GCall gname (Ctr cname cargs : args)) =
    body // zip (cvs ++ vs) (cargs ++ args) where
        (GDef _ (Pat _ cvs) vs body) = gDef p gname cname

intStep p (GCall gname (e:es)) =
    (GCall gname (intStep p e : es))

```

```
intStep p (Let (x, e1) e2) =
    e2 // [(x, e1)]
```

The interpreter `eval`, on the other hand, is a "big-step" evaluator, and hence – recursive.

```
eval :: Program -> Expr -> Expr
eval p (Ctr name args) =
    Ctr name [eval p arg | arg <- args]

eval p (FCall name args) =
    eval p (body // zip vs args) where
        (FDef _ vs body) = fDef p name

eval p (GCall gname (Ctr cname cargs : args)) =
    eval p (body // zip (cvs ++ vs) (cargs ++ args)) where
        (GDef _ (Pat _ cvs) vs body) = gDef p gname cname

eval p (GCall gname (arg:args)) =
    eval p (GCall gname (eval p arg:args))

eval p (Let (x, e1) e2) =
    eval p (e2 // [(x, e1)])
```

For the small-step interpreter it is easy to count the number of performed reduction steps – it is enough to introduce a counter in the outer loop. `intC` returns a pair `(value, n)`, where `value` is a (surprise!) value, and `n` – the number of reduction steps performed. This number is used to measure optimization "speed-up".

```
sll_run :: Task -> Env -> Value
sll_run (e, program) env = int program (e // env)

sll_trace :: Task -> Subst -> (Value, Integer)
sll_trace (e, prog) s = intC prog (e // s)

intC :: Program -> Expr -> (Expr, Integer)
intC p e = until t f (e, 0) where
    t (e, n) = isValue e
    f (e, n) = (intStep p e, n + 1)
```

From now on we start illustrating the definitions discussed with example runs. Each such example is actually a separate function definition in the module `Demonstration`. It is useful to distinguish SLL code from Haskell code. We well present SLL code in color. (Actually in `Demonstration.hs` SLL code appears as strings, which are parsed: `even(x)` ⇒ `read "even(x)"`)

The first example evaluates an expression, and also returns the number of reduction steps:

```
-- demo01
ghci> intC prog1 even(sqr(S(S(Z()))))
(True(), 15)
```

The next examples show that the small-step interpreter and the big-step interpreter give the same result:

```
-- demo02
ghci> int  prog1 even(sqr(S(S(Z()))))
True()

-- demo03
ghci> eval prog1 even(sqr(S(S(Z()))))
True()

-- demo04
ghci> int  prog1 sqr(S(S(Z())))
S(S(S(S(Z()))))

-- demo05
ghci> eval prog1 sqr(S(S(Z())))
S(S(S(S(Z()))))
```

A difference between the 2 interpreters appears, if we try to evaluate an expression with free variables (a configuration). `int` only signals an exception. `eval`, on the other hand, returns some information before signaling failure:

```
-- demo06
ghci> int  prog1 sqr(S(S(x)))
Exception: Interpreter.hs: Non-exhaustive patterns in
function intStep

-- demo07
ghci> eval  prog1 sqr(S(S(x)))
S(S( Exception: Interpreter.hs: Non-exhaustive patterns
in function eval
```

The difference is even more dramatic, if we try to evaluate an expression, which has an infinite value. `int` directly loops, while `eval` starts to produce the outermost constructors of the result.

```
ghci> prog3 = inf() = S(inf());
inf() = S(inf());
```

```
-- demo08
ghci> int prog3 inf()
^CInterrupted.

-- demo09
ghci> eval prog3 inf()
S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S
(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(
S(S(S(S(S^CInterrupted.
```

Driving.hs

`buildTree` creates a (possibly infinite) tree of configurations from a given start configuration. The main work happens inside `bt`, which proceeds recursively, maintaining not only a current configuration, but also a list of fresh names `nameSupply`. This name supply is used, when the machine `m` returns as next step `Variants [(c1, e1), (c2, e2), ...]`; in this case we recursively build the subtrees for `e1`, `e2`, ... The subtlety is that we should not use names appearing in the condition `c1` during the construction of the subtree for `e1`, as it might result in name clashes. Therefore, `(unused c ns)` is the name supply used in the corresponding recursive call.

```haskell
buildTree :: Machine Conf -> Conf -> Tree Conf
buildTree m e = bt m nameSupply e

bt :: Machine Conf -> NameSupply -> Conf -> Tree Conf
bt m ns c = case m ns c of
    Decompose ds -> Node c $ Decompose (map (bt m ns) ds)
    Transient e -> Node c $ Transient (bt m ns e)
    Stop -> Node c Stop
    Variants cs -> Node c $
        Variants [(c, bt m (unused c ns) e) | (c, e) <- cs]
```

A machine based on driving imitates a step of the execution of `eval`. `driveMachine p` creates such a machine for a given program `p`. If the configuration is a variable or a value, then the machine stops. If the configuration is a constructor with arguments, then `eval` would proceed to evaluate those arguments. As `driveMachine p` models a step of `eval`, it returns `Decompose args`, indicating that the next step should continue with evaluation of the arguments. Subexpressions `t1` and `t2` of Let-expression `let x=t1 in t2` are processed independently, as this is important for generalization: so, again we use `Decompose [t1, t2]`. Driving an indifferent-function call is simple – we return the function body with

substituted parameters (this is called "unfolding"). Driving a curious-function call, where the first argument is a constructor, proceeds in a similar way – the corresponding clause of the function definition is taken, and parameters are substituted. The most interesting cases follow: If we have a curious-function call with a first argument – a variable – then we consider all possible clauses in the definition of the function as potential continuations. This is where we use the name supply to build a new instance of the pattern with fresh names. Study the remaining last case as an exercise.

```
11 driveMachine :: Program -> Machine Conf
12 driveMachine p = drive where
13     drive :: Machine Conf
14     drive ns (Var _) = Stop
15     drive ns (Ctr _ []) = Stop
16     drive ns (Ctr _ args) = Decompose args
17     drive ns (Let (x, t1) t2) = Decompose [t1, t2]
18     drive ns (FCall name args) =
19         Transient $ e // (zip vs args) where
20             FDef _ vs e = fDef p name
21     drive ns (GCall gn (Ctr cn cargs : args)) =
22         Transient $ e // sub where
23             (GDef _ (Pat _ cvs) vs e) = gDef p gn cn
24             sub = zip (cvs ++ vs) (cargs ++ args)
25     drive ns (GCall gn args@((Var _):_)) =
26         Variants $ variants gn args where
27             variants gn args =
28                 map (scrutinize ns args) (gDefs p gn)
29     drive ns (GCall gn (inner:args)) =
30         inject (drive ns inner) where
31             inject (Transient t) =
32                 Transient (GCall gn (t:args))
33             inject (Variants cs) = Variants $ map f cs
34             f (c, t) = (c, GCall gn (t:args))
35
36 scrutinize :: NameSupply -> [Expr] -> GDef
37     -> (Contract, Expr)
38 scrutinize ns (Var v : args)
39     (GDef _ (Pat cn cvs) vs body) =
40     (Contract v (Pat cn fresh), body // sub) where
41         fresh = take (length cvs) ns
42         sub = zip (cvs ++ vs) (map Var fresh ++ args)
```

Some examples follow – a driving step with case analysis:

```
-- demo10
ghci> driveMachine prog1 nameSupply
        odd(add(x, mult(x, S(x))))
x == Z() => odd(mult(x, S(x)))
x == S(v1) => odd(S(add(v1, mult(x, S(x)))))
```

A transient step:

```
-- demo11
ghci> driveMachine prog1 nameSupply
        odd(S(add(v1, mult(x, S(x)))))
=> even(add(v1, mult(x, S(x))))
```

An infinite tree of configurations:

```
-- demo12
ghci> buildTree (driveMachine prog1) even(sqr(x))
|__even(sqr(x))
 |
 |__even(mult(x, x))
  ?x == Z()
  |__even(Z())
   |
   |__True()
  ?x == S(v1)
  |__even(add(x, mult(v1, x)))
   ?x == Z()
   |__even(mult(v1, x))
    ?v1 == Z()
    |__even(Z())
     |
     |__True()
    ?v1 == S(v2)
    |__even(add(x, mult(v2, x)))
     ?x == Z()
     |__even(mult(v2, x))
      ?v2 == Z()
      |__even(Z())
       |
       |__True()
      ?v2 == S(v3)
      |__even(add(x, mult(v3, x)))
  ...
```

TreeInterpreter.hs

The tree interpreter contains not so many surprises. The only interesting cases are:

- a node with variants (`Node e (Variants cs)`): we choose that subbranch, whose condition matches the current environment (`env`);
- a folding node (`Node _ (Fold t ren)`): here we continue with the subtree (which would correspond to a loop in the graph), first performing the corresponding renaming of the current environment.

```
1  intTree :: Tree Conf -> Env -> Value
2  intTree (Node e Stop) env =
3      e // env
4  intTree (Node (Ctr cname _) (Decompose ts)) env =
5      Ctr cname $ map (\t -> intTree t env) ts
6  intTree (Node _ (Transient t)) env =
7      intTree t env
8  intTree (Node e (Variants cs)) env =
9      head $ catMaybes $ map (try env) cs
10 intTree (Node (Let (v, e1) e2) (Decompose [t1, t2])) env =
11     intTree t2 ((v, intTree t1 env) : env)
12 intTree (Node _ (Fold t ren)) env =
13     intTree t $ map (\(k, v) -> (renKey k, v)) env where
14         renKey k = maybe k fst (find ((k ==) . snd)  ren)
15
16 try :: Env -> (Contract, Tree Conf) -> (Maybe Expr)
17 try env (Contract v (Pat pn vs), t) =
18     if cn == pn then (Just $ intTree t extEnv)
19     else Nothing where
20         c@(Ctr cn cargs) = (Var v) // env
21         extEnv = zip vs cargs ++ env
```

An example of using infinite trees for evaluation:

```
-- demo13
ghci> intTree (buildTree (driveMachine prog1) even(fSqr(x)))
         [("x", S(S(Z())))]
True()

-- demo14
ghci> intTree (buildTree (driveMachine prog1) even(fSqr(x)))
         [("x", S(S(S(Z()))))]
False()
```

Folding.hs

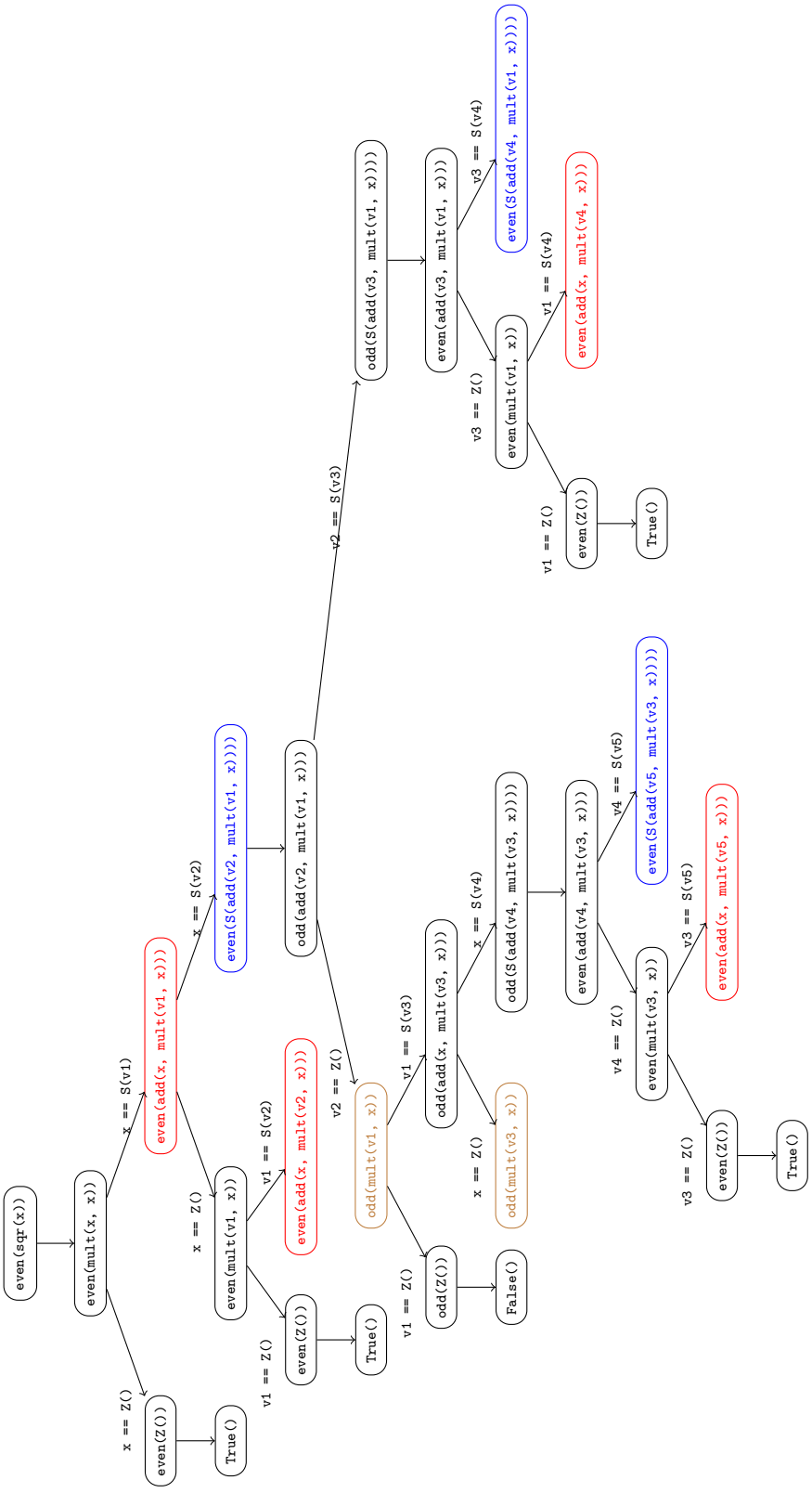
The tree of configurations is folded into a graph using the well-known technique of "tying the knot" (`http://www.haskell.org/haskellwiki/Tying_the_Knot`). We traverse the tree top-down, accumulating the encountered configurations. If the current configuration coincides with a previous one modulo renaming, then a loop is formed and we do not go inside the corresponding subtree.

```
foldTree :: Tree Conf -> Graph Conf
foldTree t = fixTree (tieKnot []) t

tieKnot :: [Node Conf] -> Node Conf -> Tree Conf
    -> Graph Conf
tieKnot ns n t@(Node e _) =
    case [(k, r) | k <- n:ns, isCall e,
              Just r <- [renaming (nodeLabel k) e]] of
        [] -> fixTree (tieKnot (n:ns)) t
        (k, r):_ -> Node e (Fold k r)

fixTree :: (Node t -> Tree t -> Graph t) -> Tree t
    -> Graph t
fixTree f (Node e (Transient c)) = t where
    t = Node e $ Transient $ f t c
fixTree f (Node e (Decompose cs)) = t where
    t = Node e $ Decompose [f t c | c <- cs]
fixTree f (Node e (Variants cs)) = t where
    t = Node e $ Variants [(p, f t c) | (p, c) <- cs]
fixTree f (Node e Stop) = (Node e Stop)
```

The following example is shown in the main article, subsection "Folding".

```
-- demo15
ghci> foldTree $ buildTree (driveMachine prog1)
    even(sqr(x))
...
```

We repeat the graph drawing here, marking coinciding (up to renaming) nodes with the same color.


even(sqr(x))
even(mult(x, x))
x == Z()
x == S(v1)
even(Z())
True()
even(add(x, mult(v1, x)))
even(mult(v1, x))
x == Z()
x == S(v2)
v1 == Z()
v1 == S(v2)
even(Z())
True()
even(add(x, mult(v2, x)))
even(S(add(v2, mult(v1, x))))
odd(add(v2, mult(v1, x)))
v2 == Z()
v2 == S(v3)
odd(mult(v1, x))
v1 == Z()
v1 == S(v3)
odd(Z())
False()
odd(add(x, mult(v3, x)))
x == Z()
x == S(v4)
odd(mult(v3, x))
odd(S(add(v3, mult(v1, x))))
odd(S(add(v4, mult(v3, x))))
even(add(v4, mult(v3, x)))
v4 == Z()
v4 == S(v5)
even(mult(v3, x))
v3 == Z()
v3 == S(v5)
even(Z())
True()
even(S(add(v5, mult(v3, x))))
even(add(x, mult(v5, x)))
even(add(v3, mult(v1, x)))
even(mult(v1, x))
v3 == Z()
v3 == S(v4)
v1 == Z()
v1 == S(v4)
even(Z())
True()
even(S(add(v4, mult(v1, x))))
even(add(x, mult(v4, x)))

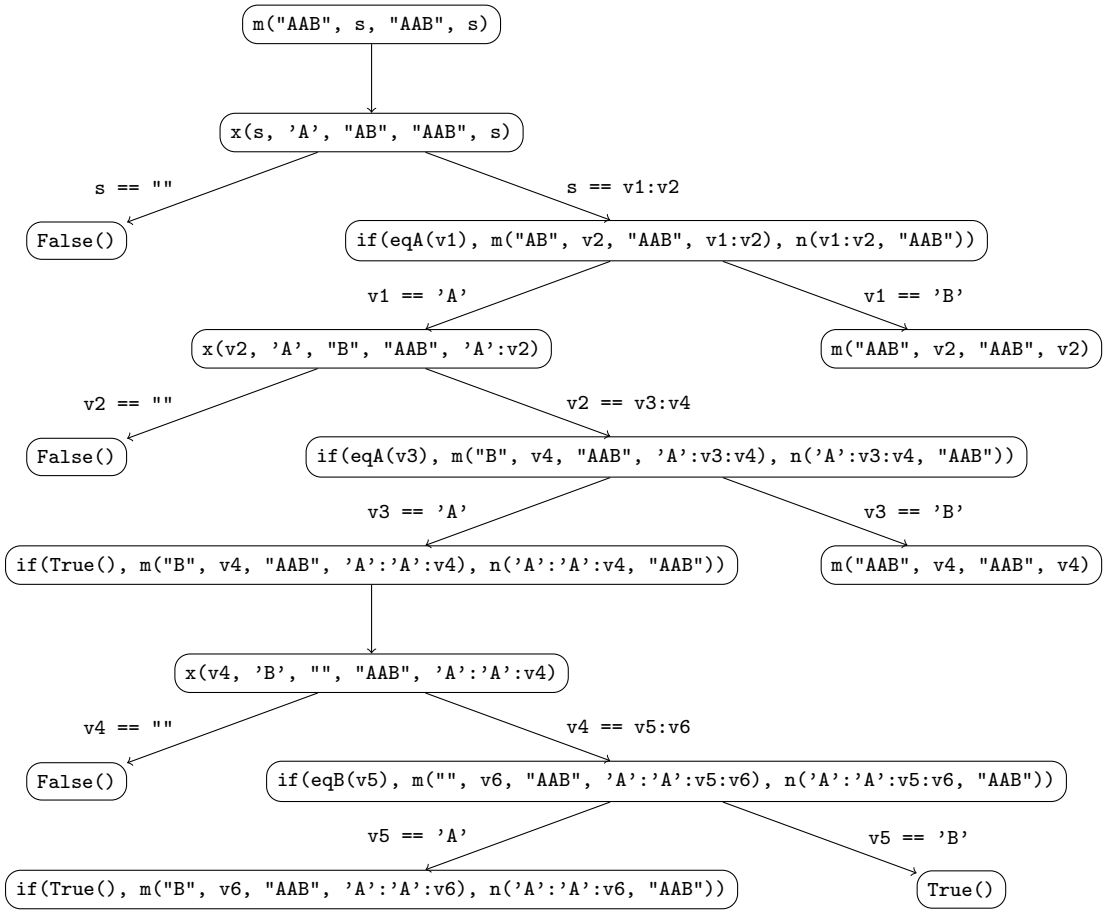
Generator.hs

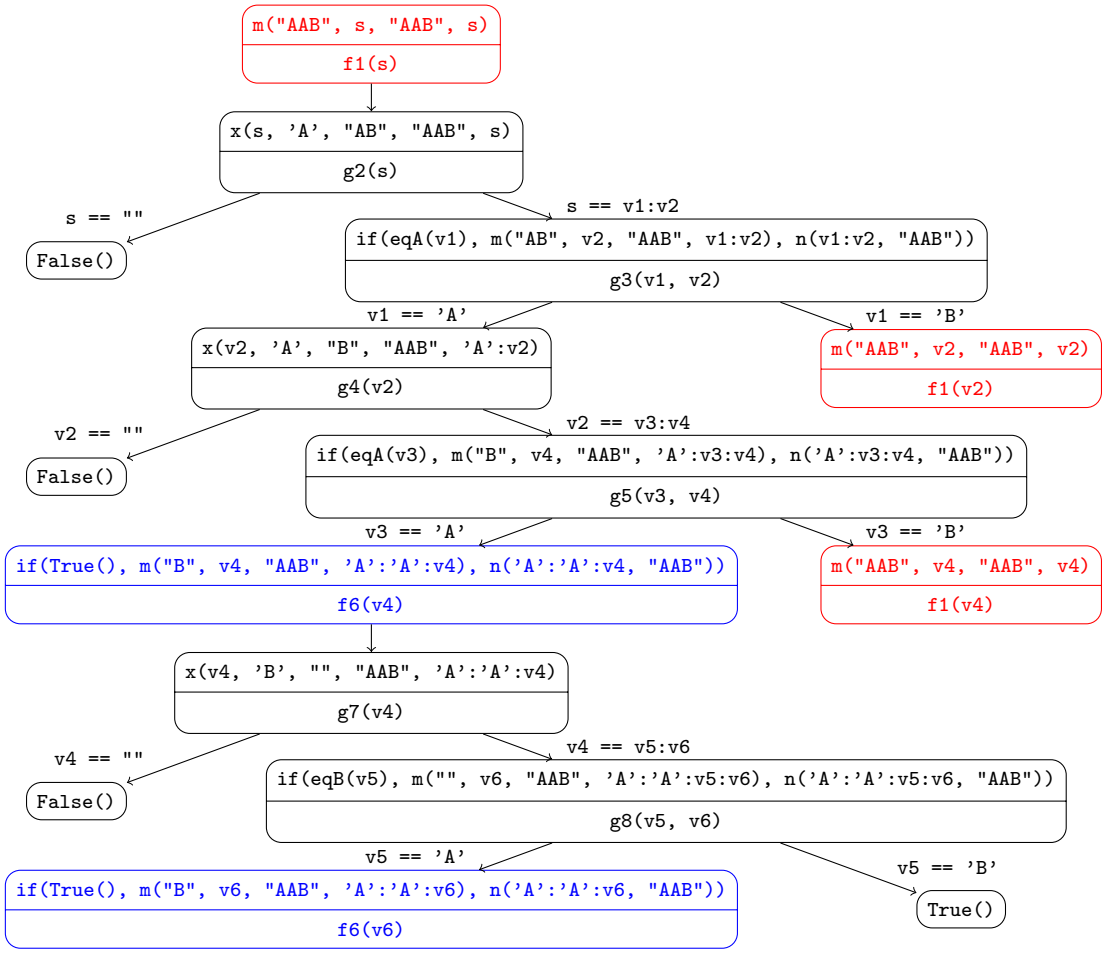
The module `Generator` is quite big, and full of technical details, mostly related to working with names

```
residuate :: Graph Conf -> Task
residuate tree = (expr, program) where
    (expr, program, _) = res nameSupply [] tree

res :: NameSupply -> [(Conf, Conf)] -> Graph Conf
    -> (Conf, Program, NameSupply)
res ns mp (Node e Stop) = (e, Program [] [], ns)

res ns mp (Node (Ctr cname _) (Decompose ts)) =
    (Ctr cname args, p1, ns1) where
        (args, p1, ns1) = res' ns mp ts

res ns mp (Node (Let (v, _) _) (Decompose ts)) =
    (e2 // [(v, e1)], p1, ns1) where
        ([e1, e2], p1, ns1) = res' ns mp ts

res (n:ns) mp (Node e (Transient t)) =
    (fcall, Program ((FDef f1 vs body):fs) gs, ns1) where
        vs = vnames e
        f1 = "f" ++ (tail n)
        fcall = FCall f1 $ map Var vs
        (body, Program fs gs, ns1) =
            res ns ((e, fcall) : mp) t

res (n:ns) mp (Node e (Variants cs)) =
    (gcall, Program fs (newGs ++ gs), ns1) where
        vs@(pv:vs') = vnames e
        (vs_, vs'_) =
            if (isRepeated pv e) && (isUsed pv cs)
            then (pv:vs, vs) else (vs, vs')
        g1 = "g" ++ (tail n)
        gcall = GCall g1 $ map Var vs_
        (bodies, Program fs gs, ns1) =
            res' ns ((e, gcall) : mp) $ map snd cs
        pats = [pat | (Contract v pat, _) <- cs]
        newGs = [GDef g1 p vs'_ b |
                 (p, b) <-  (zip pats bodies)]
        isUsed vname cs =
            any (any (== vname) . vnames . nodeLabel . snd)
```

```
40                  cs
41
42  res ns mp (Node e (Fold (Node base _) ren)) =
43      (call, Program [] [], ns) where
44          call = baseCall // [(x, Var y) | (x, y) <- ren]
45          Just baseCall = lookup base mp
46
47  res' :: NameSupply -> [(Conf, Conf)] -> [Graph Conf]
48      -> ([Conf], Program, NameSupply)
49  res' ns mp ts = foldl f ([], Program [] [], ns) ts where
50      f (cs, Program fs gs, ns1) t =
51          (cs ++ [g], Program (fs ++ fs1) (gs ++ gs1), ns2)
52          where
53              (g, Program fs1 gs1, ns2) = res ns1 mp t
54
55  isBase e1 (Node _ (Decompose ts)) =
56      or $ map (isBase e1) ts
57  isBase e1 (Node _ (Variants cs)) =
58      or $ map (isBase e1 . snd) cs
59  isBase e1 (Node _ (Transient t)) = isBase e1 t
60  isBase e1 (Node _ (Fold (Node e2 _) _)) = e1 == e2
61  isBase e1 (Node e2 Stop) = False
```

`residuate` delegates the main part of the work to `res`, which processes the tree top-down, left-to-right. The result of traversing each subtree is a new configuration and a program (a list of indifferent and curious function definitions). The main complication is to ensure a unique name for each generated function. A more detailed explanation would probably get too long. The reader is rather invited to study directly the sources. To aid understanding, we list the graph obtained for the KMP-test from the main article, and then repeated the same graph overlaid with the new generated configurations.


m("AAB", s, "AAB", s)
x(s, 'A', "AB", "AAB", s)
s == ""
s == v1:v2
False()
if(eqA(v1), m("AB", v2, "AAB", v1:v2), n(v1:v2, "AAB"))
v1 == 'A'
v1 == 'B'
x(v2, 'A', "B", "AAB", 'A':v2)
m("AAB", v2, "AAB", v2)
v2 == ""
v2 == v3:v4
False()
if(eqA(v3), m("B", v4, "AAB", 'A':v3:v4), n('A':v3:v4, "AAB"))
v3 == 'A'
v3 == 'B'
if(True(), m("B", v4, "AAB", 'A':'A':v4), n('A':'A':v4, "AAB"))
m("AAB", v4, "AAB", v4)
x(v4, 'B', "", "AAB", 'A':'A':v4)
v4 == ""
v4 == v5:v6
False()
if(eqB(v5), m("", v6, "AAB", 'A':'A':v5:v6), n('A':'A':v5:v6, "AAB"))
v5 == 'A'
v5 == 'B'
if(True(), m("B", v6, "AAB", 'A':'A':v6), n('A':'A':v6, "AAB"))
True()



In the last figure we again mark by the same color nodes, which are equal up to renaming. The upper part of each node contains the original configuration, the lower part – the new configuration generated by `res`.

```
ghci> let g = ...

-- demo24
ghci> residuate g
f1(s)
f1(s) = g2(s);
g2("") = False();
g2(v1:v2) = g3(v1, v2);
g3('A', v2) = g4(v2);
g3('B', v2) = f1(v2);
g4("") = False();
g4(v3:v4) = g5(v3, v4);
```

```
g5('A', v4) = f6(v4);
g5('B', v4) = f1(v4);
f6(v4) = g7(v4);
g7("") = False();
g7(v5:v6) = g8(v5, v6);
g8('A', v6) = f6(v6);
g8('B', v6) = True();
```

Propotype.hs

This is the simplest "prototype" supercompiler. No simplification (transient step removal) of the graph is performed, no information propagation is done.

```
1  transform :: Task -> Task
2  transform (e, p) =
3      residuate $ foldTree $ buildFTree (driveMachine p) e
```

`buildFTree` builds the foldable tree. The only difference with `buildTree` lies in the fact, that generalization is performed if the whistle blows:

```
4  buildFTree :: Machine Conf -> Conf -> Tree Conf
5  buildFTree m e = bft m nameSupply e
6
7  bft :: Machine Conf -> NameSupply -> Conf -> Tree Conf
8  bft d (n:ns) e | whistle e = bft d ns $ generalize n e
9  bft d ns     t | otherwise = case d ns t of
10     Decompose ds -> Node t $ Decompose $ map (bft d ns) ds
11     Transient e -> Node t $ Transient $ bft d ns e
12     Stop -> Node t Stop
13     Variants cs -> Node t $
14         Variants [(c, bft d (unused c ns) e) | (c, e) <- cs]
```

Probably the simplest possible whistle:

```
15 sizeBound = 40
16 whistle :: Expr -> Bool
17 whistle e@(FCall _ args) =
18     not (all isVar args) && size e > sizeBound
19 whistle e@(GCall _ args) =
20     not (all isVar args) && size e > sizeBound
21 whistle _ = False
```

Generalization is also extremely simple, compared to most other supercompilers – the biggest subexpression is lifted into a `let` for separate processing:

```
generalize :: Name -> Expr -> Expr
generalize n (FCall f es) =
    Let (n, e) (FCall f es') where
        (e, es') = extractArg n es
generalize n (GCall g es) =
    Let (n, e) (GCall g es') where
        (e, es') = extractArg n es

extractArg :: Name -> [Expr] -> (Expr, [Expr])
extractArg n es = (maxE, vs ++ Var n : ws) where
    maxE = maximumBy ecompare es
    ecompare x y =
        compare (eType x * size x) (eType y * size y)
    (vs, w : ws) = break (maxE ==) es
    eType e = if isVar e then 0 else 1
```

Deforester.hs

If we just add `simplify` (transient edge removal) to the text of `transform`, we obtain deforestation:

```
deforest :: Task -> Task
deforest (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (driveMachine p) e

simplify :: Graph Conf -> Graph Conf
simplify (Node e (Decompose ts)) =
    Node e (Decompose $ map simplify ts)
simplify (Node e (Variants cs)) =
    Node e (Variants [(c, simplify t) | (c, t) <- cs])
simplify (Node e (Transient t)) | isBase e t =
    Node e $ Transient $ simplify t
simplify (Node e (Transient t)) =
    simplify t
simplify t = t
```

Supercompiler.hs

As a next step, we extend deforestation with information propagation, to obtain our final supercompiler:

```
supercompile :: Task -> Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (addPropagation $ driveMachine p) e

addPropagation :: Machine Conf -> Machine Conf
addPropagation m ns e = propagateContract (m ns e)

propagateContract :: Step Conf -> Step Conf
propagateContract (Variants vs) =
  Variants [(c, e // [(v, Ctr cn $ map Var vs)]) |
            (c@(Contract v (Pat cn vs)), e) <- vs]
propagateContract step = step
```

The reader is encouraged to compare the trees produced by different transformers:

```
-- demo21
ghci> foldTree $ buildFTree (driveMachine prog2)
    match("AAB", s)
...

-- demo22
ghci> simplify $ foldTree $ buildFTree (driveMachine prog2)
    match("AAB", s)
...

-- demo23
ghci> simplify $ foldTree $ buildFTree (addPropagation
(driveMachine prog2)) conf2
...
```

...and also the resulting residual tasks (listed also in the main article):

```
-- demo24
ghci> transform (match("AAB", s), prog2)
...

-- demo25
ghci> deforest (match("AAB", s), prog2)
...
```

```
-- demo26
ghci> supercompile (match("AAB", s), prog2)
...
```

Anti-KMP test

This section lists in detail the results of the anti-KMP-test mentioned in the main text. To improve readability, we set specifically for these examples `sizeBound=10`.

The baseline transformer produces 19 functions.

```
-- demo18
ghci> transform (even(sqr(x)), prog1)
f1(x)
f1(x) = g2(x, x);
f3() = True();
f6() = True();
f7(v2, v1, x) = g8(v2, v1, x);
f10() = False();
f12(v4, v3, x) = g13(v4, v3, x);
f15() = True();
f16(v3, v1, x) = g17(v3, v1, x);
f19() = True();
g2(Z(), x) = f3();
g2(S(v1), x) = g4(x, x, v1);
g4(Z(), x, v1) = g5(v1, x);
g4(S(v2), x, v1) = f7(v2, v1, x);
g5(Z(), x) = f6();
g5(S(v2), x) = g4(x, x, v2);
g8(Z(), v1, x) = g9(v1, x);
g8(S(v3), v1, x) = f16(v3, v1, x);
g9(Z(), x) = f10();
g9(S(v3), x) = g11(x, x, v3);
g11(Z(), x, v3) = g9(v3, x);
g11(S(v4), x, v3) = f12(v4, v3, x);
g13(Z(), v3, x) = g14(v3, x);
g13(S(v5), v3, x) = f7(v5, v3, x);
g14(Z(), x) = f15();
g14(S(v5), x) = g4(x, x, v5);
g17(Z(), v1, x) = g18(v1, x);
g17(S(v4), v1, x) = f7(v4, v1, x);
g18(Z(), x) = f19();
g18(S(v4), x) = g4(x, x, v4);
```

Deforestation – 11 functions:

```
-- demo19
ghci> deforest (even(sqr(x)), prog1)
g1(x, x)
f4(v2, v1, x) = g5(v2, v1, x);
g1(Z(), x) = True();
g1(S(v1), x) = g2(x, x, v1);
g2(Z(), x, v1) = g3(v1, x);
g2(S(v2), x, v1) = f4(v2, v1, x);
g3(Z(), x) = True();
g3(S(v2), x) = g2(x, x, v2);
g5(Z(), v1, x) = g6(v1, x);
g5(S(v3), v1, x) = g10(v3, v1, x);
g6(Z(), x) = False();
g6(S(v3), x) = g7(x, x, v3);
g7(Z(), x, v3) = g6(v3, x);
g7(S(v4), x, v3) = g8(v4, v3, x);
g8(Z(), v3, x) = g9(v3, x);
g8(S(v5), v3, x) = f4(v5, v3, x);
g9(Z(), x) = True();
g9(S(v5), x) = g2(x, x, v5);
g10(Z(), v1, x) = g11(v1, x);
g10(S(v4), v1, x) = f4(v4, v1, x);
g11(Z(), x) = True();
g11(S(v4), x) = g2(x, x, v4);
```

Supercompilation – 34 functions.

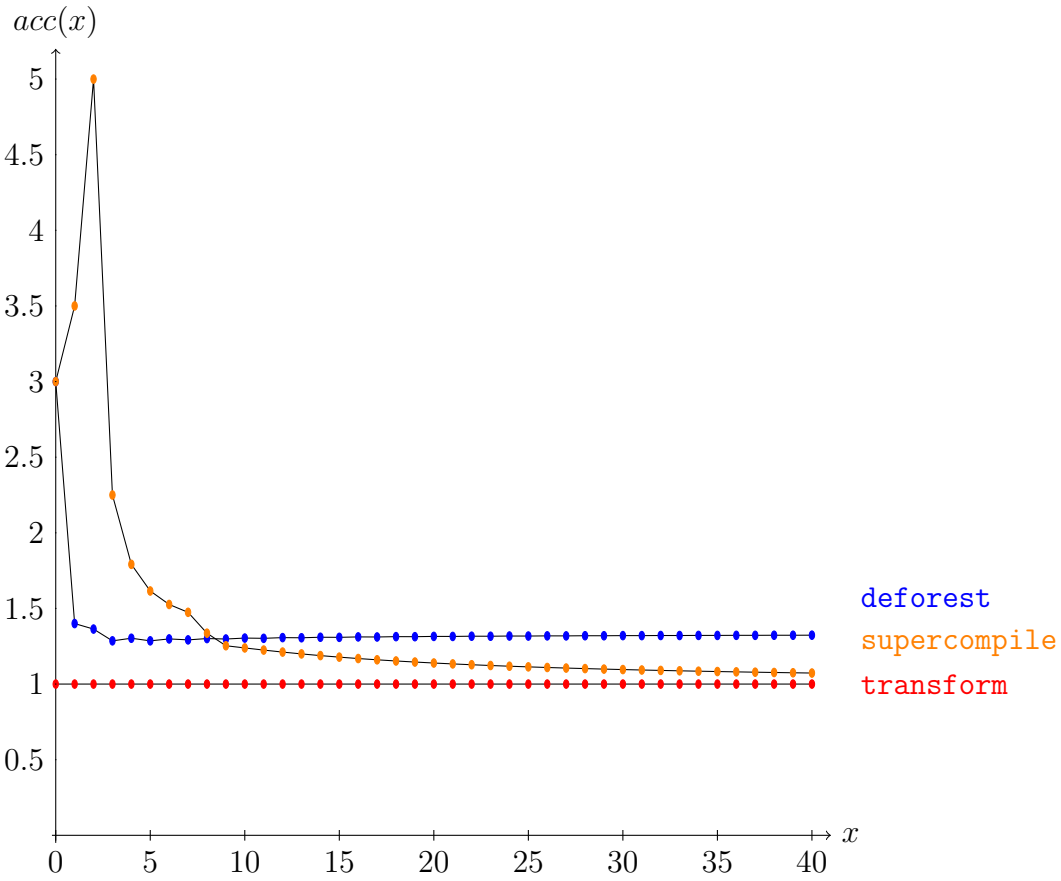
```
-- demo20
ghci> supercompile (even(sqr(x)), prog1)
g1(x)
g1(Z()) = True();
g1(S(v1)) = g2(v1);
g2(Z()) = False();
g2(S(v2)) = g3(v2);
g3(Z()) = True();
g3(S(v3)) = g33(S(g4(v3)));
g4(Z()) = S(S(S(S(S(S(Z()))))));
g4(S(v5)) = S(g32(v5, S(S(S(S(g31(v5, S(S(S(S(g30(v5,
    S(S(S(S(g29(v5, g5(v5))))))))))))))));
g5(Z()) = Z();
g5(S(v10)) = S(S(S(S(S(g28(v10, g6(v10)))))));
g6(Z()) = Z();
g6(S(v12)) = S(S(S(S(S(S(g27(v12, g7(v12)))))))));
```

```
g7(Z()) = Z();
g7(S(v14)) = S(S(S(S(S(S(S(g26(v14, g8(v14)))))))));
g8(Z()) = Z();
g8(S(v16)) = g25(S(S(S(S(S(S(S(S(v16)))))))),
    g9(v16, S(S(S(S(S(S(S(S(v16)))))))));
g9(Z(), v18) = Z();
g9(S(v19), v18) = g10(v18, v19);
g10(Z(), v19) = g11(v19);
g10(S(v20), v19) = S(g12(v20, v19));
g11(Z()) = Z();
g11(S(v20)) = g11(v20);
g12(Z(), v19) = g13(v19);
g12(S(v21), v19) = S(g14(v21, v19));
g13(Z()) = Z();
g13(S(v21)) = S(g13(v21));
g14(Z(), v19) = g15(v19);
g14(S(v22), v19) = S(g16(v22, v19));
g15(Z()) = Z();
g15(S(v22)) = S(S(g15(v22)));
g16(Z(), v19) = g17(v19);
g16(S(v23), v19) = S(g18(v23, v19));
g17(Z()) = Z();
g17(S(v23)) = S(S(S(g17(v23))));
g18(Z(), v19) = g19(v19);
g18(S(v24), v19) = S(g20(v24, v19));
g19(Z()) = Z();
g19(S(v24)) = S(S(S(S(g19(v24)))));
g20(Z(), v19) = g21(v19);
g20(S(v25), v19) = S(g24(v25, g22(v19, v25)));
g21(Z()) = Z();
g21(S(v25)) = S(S(S(S(S(g21(v25))))));
g22(Z(), v25) = Z();
g22(S(v27), v25) = S(S(S(S(S(S(g23(v25, g22(v27,
    v25)))))))));
g23(Z(), v28) = v28;
g23(S(v29), v28) = S(g23(v29, v28));
g24(Z(), v26) = v26;
g24(S(v27), v26) = S(g24(v27, v26));
g25(Z(), v17) = v17;
g25(S(v19), v17) = S(g25(v19, v17));
g26(Z(), v15) = v15;
g26(S(v16), v15) = S(g26(v16, v15));
g27(Z(), v13) = v13;
```

```
g27(S(v14), v13) = S(g27(v14, v13));
g28(Z(), v11) = v11;
g28(S(v12), v11) = S(g28(v12, v11));
g29(Z(), v9) = v9;
g29(S(v10), v9) = S(g29(v10, v9));
g30(Z(), v8) = v8;
g30(S(v9), v8) = S(g30(v9, v8));
g31(Z(), v7) = v7;
g31(S(v8), v7) = S(g31(v8, v7));
g32(Z(), v6) = v6;
g32(S(v7), v6) = S(g32(v7, v6));
g33(Z()) = True();
g33(S(v5)) = g34(v5);
g34(Z()) = False();
g34(S(v6)) = g33(v6);
```

We compare below the speed-ups of the task `even(sqr(x))`, `prog1` obtained by using `transform`, `deforest`, and `supercompile`. The speed-up of task t_2 with respect to task t_1 is calculated as follows (where `x` is a natural number, and `x` – the corresponding Peano number):

```
acc t1 t2 x = steps1 / steps2 where
    (_, steps1) = sll_trace t1 [("x"), x]
    (_, steps2) = sll_trace t2 [("x"), x]
```



These results demonstrate, that the speed-up from supercompilation is most impressive for small x; for such small numbers we do not reach the cases of generalization present in the graph. When grows, however, the drawbacks of performing generalization start to show, and the deforested task turns out faster.