Towards Understanding Superlinear Speedup by Distillation

Neil D. Jones

Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
neil@diku.dk

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
hamilton@computing.dcu.ie

Abstract. Distillation is a fully automatic program transformation that can yield superlinear program speedups. Bisimulation is a key to the proof that distillation is correct, i.e., preserves semantics. Bisimulation normally requires explicit definition of equivalent states. However distillation can produce complexity reductions and thus fewer residual states. This often makes 1-1 relations between program states of original and transformed programs hard or impossible to see. The correctness proof of distillation, since based on observational equivalence, is insensitive to program running times, and does not help to explain how superlinear speedups can occur. This paper's approach to better understanding cause-and-effect in distillation is to simplify distillation as much as possible, while maintaining its capacity for superlinear speedups. We show *how distillation can give superlinear speedups* on some "old chestnut" programs well-known from the early program transformation literature: naive reverse, factorial sum, Fibonacci, and palindrome detection. We describe current work on such questions, partly theoretical and partly computer experiments. Furthermore, we show using complexity-theoretic tools that a sizable class of exponential-time programs can be converted into second-order polynomial-time equivalents. The idea is to trade time for space, in effect replacing `cons` or a Turing machine tape by first-order functions as arguments in a `cons`-free program. Finally, we conjecture that distillation can realise these superlinear speedup transformations in general.

1 Introduction

Distillation, supercompilation, and partial evaluation are automatic program transformations (see [11–13, 21–23]). The main goal of all three is to transform a program into an improved program. Partial evaluation has been fairly well automated [13]. A breakthrough occurred when the *Futamura projections* (Futamura, Ershov [7, 8]) were realised in practice: generating a compiler from an interpreter by self-applying a partial evaluator (see [13] for details and history). Furthermore, optimal specialisation has been achieved: partial evaluation can remove all interpretation overhead when specialising an interpreter to its program input.

In some respects supercompilation, deforestation and distillation (Turchin, Sørensen, Wadler, Hamilton [10–13,21,23,24]) can make deeper transformations on program control structure. A well-known example is that deforestation can transform a multipass program into a single pass algorithm [23,24], a feat beyond the reach of current partial evaluators.

1.1 Goal: extend automatic superlinear program speedup

Program optimisations by hand (Burstall-Darlington and many others [1, 3]) sometimes yield superlinear program speedups. Transformation can make substantial improvements, for instance changing a program running in time $O(n^2)$ or even $O(2^n)$ into one running in time $O(n)$. Familiar examples include naive programs for list reversal, sum of factorials, and the Fibonacci function. A goal for many years now has been how to obtain such effects *by well-automated methods.*

Classical compiler optimisations are a model of automation, though the program speedups they give are limited. Many have been proven correct using bisimulation, e.g., [17] by Lacey et al. This has led to some practical automation of compiler correctness proofs, e.g., [18] by Lerner et al, and successors.

However it has been proven (see [13,21]) that partial evaluation, deforestation and supercompilation (as well as most classical compiler optimisations) are all limited to *at most program speedups by linear constant factors.* One reason for such limited optimisation speedups is that the bisimulations of [17] all involve *one-to-one relations* between the control points of the original program and the compiler-optimised program.

In contrast, distillation [10, 11]) can yield superlinear asymptotic speedups: this refinement of supercompilation can sometimes transform a program into a semantically equivalent but asymptotically faster equivalent.

1.2 Bisimulation and program transformation.

Correctness of transformation can be proven using bisimulation [11, 12, 17] to relate computations by the original and the transformed programs. A question:

> How can a program running in time $O(n^2)$ (or even time $O(2^n)$) be bisimilar to a program running in time $O(n)$?

This puzzling question was the starting point of this work. It was clear at once that *1-1 relations between program control points would not suffice to explain the phenomenon.* A challenge to overcome: the system structure and techniques used in distillation as in [10–12] are complex, and hard to reason about globally.

This paper's approach to better understanding cause-and-effect in distillation is to *simplify distillation as much as possible*, while maintaining its capacity for superlinear speedups. We will describe current work on such questions, partly theoretical and partly computer experiments.

2 A language, observational equivalence, and labeled transition systems

Our approach is to simplify the general distillation techniques of [10–12], so its essence can be seen in a more limited context, to see what is happening abstractly. A clearer understanding of cause-and-effect could show how automatically to achieve superlinear speedup on a wider range of programs.

A longer-term goal is to apply distillation techniques to intermediate program representations in a compiler, where programs are call-by-value and imperative, i.e., tail-recursive. Related: Debois applies partial evaluation to realise some optimisations of intermediate program representations in a compiler [6]; however, superlinear speedup is beyond the current state of the art.

2.1 Source language syntax

Data: let Σ be an uninterpreted signature for constructors, and let T_Σ be the set of all well-formed trees over Σ, finite or infinite. Our examples use as constructors 0-ary 0, unary 1+ (successor), and binary constructors $+, *, ::$. The net effect of a program is to compute a (mathematical, partial) function $f : (T_\Sigma)^n \rightharpoonup T_\Sigma$.

Programs are first-order, built from variables x, constructors c, function calls, and **case**. Calls and constructor applications must have all their arguments, i.e., full arities. Semantics $[\![prog]\!] : (T_\Sigma)^n \rightharpoonup T_\Sigma$ is call-by-value, omitted for brevity and because of familiarity.

$prog$	$::= e$ **where** Δ	
Δ	$::= f_1\ x_1 \ldots x_n = e_1 \quad \ldots \quad f_m\ x_1 \ldots x_p = e_m$	Function definitions
e	$::= x \mid c\ e_1 \ldots e_k \mid call \mid case$	Expression
$call$	$::= f\ e_1 \ldots e_n$	Function call
$case$	$::=$ **case** e **of** $p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$	Case expression
p	$::= c\ x_1 \ldots x_k$	Case pattern

Free variables are allowed only in the e part of program e **where** Δ. All other variables must be bound, either by function parameters or in case patterns.

Definition 1. *Denote by* $time_p(x) \in \mathbb{N} \cup \{\infty\}$ *the running time of program* p *on input* x*, e.g., the number of steps used in computing* $[\![p]\!](x)$.

Goal: automatically transform program p into program p' such that $[\![p]\!] = [\![p']\!]$, but $time_{p'} < time_p$ asymptotically, i.e., in the limit as input size grows.

2.2 Observational equivalence and labeled transition systems

Distillation transforms a program p_1 into an observationally equivalent program p_2. (Two central references: Milner and Gordon [9, 19].) Observational equivalence implies semantic equivalence, i.e., $p_1 \simeq p_2$ implies $[\![p_1]\!] = [\![p_2]\!]$.

For definitions of context $C[]$ and evaluation $\Downarrow$ appropriate to call-by-value:

Definition 2 (Observational Equivalence). Programs p_1, p_2 are *observationally equivalent*, written $p_1 \simeq p_2$, if and only if they have the same termination behaviour in all closing contexts, i.e., $p_1 \simeq p_2$ iff $\forall C[] \,.\, C[p_1] \Downarrow$ iff $C[p_2] \Downarrow$.

A limitation of observational equivalence Unfortunately (from this paper's perspective), observational equivalence $p \simeq p'$ tells us *nothing whatsoever* about the comparative running times of the programs involved. In each instance of our "old chestnut" programs, the original program is observationally equivalent to its optimised version. Our goal: to clarify the relation between program running times before and after distillation.

Definition 3 (Labeled transition systems). *A labeled transition system (LTS for short) is a tuple* $t = (\mathcal{S}, s_0, \rightarrow, Act)$ *where* $\mathcal{S}$ *is a set of* states. $s_0 \in \mathcal{S}$ *is the* root state, **0** *is the end-of-action state, and Act is a set of actions* α. *The* transition relation *is* $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$. Notation: *as usual we write a transition* (s, α, s') *in* $\rightarrow$ *as* $s \xrightarrow{\alpha} s'$.

Definition 4 (Simulation). Binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a *simulation* of LTS $t_1 = (\mathcal{S}_1, s_0^1, \rightarrow_1, Act)$ by LTS $t_2 = (\mathcal{S}_2, s_0^2, \rightarrow_2, Act)$ if $(s_0^1, s_0^2) \in \mathcal{R}$, and for every pair $(s_1, s_2) \in \mathcal{R}$ and $\alpha \in Act, s_1' \in \mathcal{S}_1$:

$$\text{if } s_1 \xrightarrow{\alpha} s_1' \text{ then } \exists s_2' \in \mathcal{S}_2 \,.\, s_2 \xrightarrow{\alpha} s_2' \wedge (s_1', s_2') \in \mathcal{R}$$

Note: t_1 and t_2 must have the same action sets.

Definition 5 (Bisimulation). A *bisimulation* $\sim$ is a binary relation $\mathcal{R}$ such that both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are simulations.

Using an LTS as a program's abstract syntax. Represent a variable x by a transition $s \xrightarrow{x} \mathbf{0}$; represent $c\ e_1 \ldots e_k$ where c is a constructor by transitions $s \xrightarrow{c} \mathbf{0}, s \xrightarrow{\#1} s_1, \ldots, s \xrightarrow{\#k} s_k$, where s_i is the root of the LTS representation of expression e_i; represent **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ by transitions $s \xrightarrow{\textbf{case}} s_0, s \xrightarrow{p_1} s_1, \ldots, s \xrightarrow{p_k} s_k$; and represent a function call $f\ e_1 \ldots e_n$ by transitions $s \xrightarrow{\textbf{call}} s_0, s \xrightarrow{x_1} s_1, \ldots, s \xrightarrow{x_n} s_n$ where Δ contains function definition $f\ x_1 \ldots x_n = e_0$.

2.3 Example: "naive reverse" program representation as an LTS

```
nr input where

nr xs = case xs of
            nil       => nil
          | (:: y ys) => (ap (nr ys) (:: y nil))

ap us vs = case us of
               nil        => vs
             | (:: u us1) => (:: u (ap us1 vs))
```

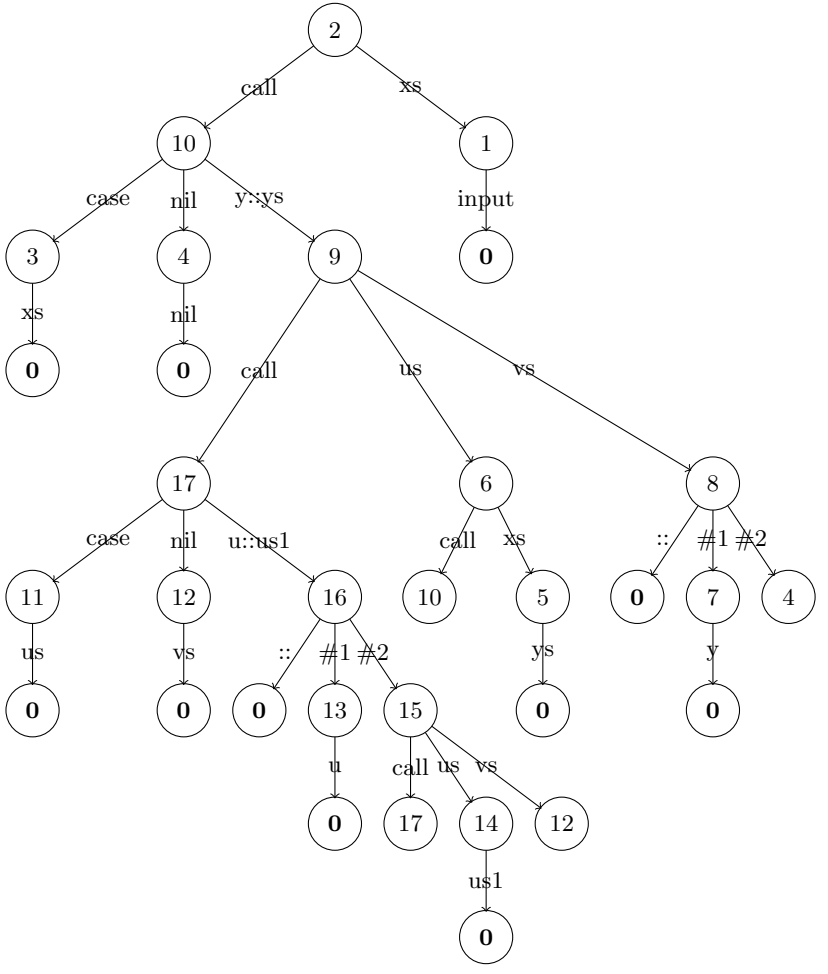

Fig. 1. Labelled Transition System for "naive reverse" program

The LTS representation of the naive reverse program is diagrammed in Fig. 1. This may be easier to follow than an unstructured set of transitions such as

$$\{\ 2 \xrightarrow{\textbf{call}} 10,\ 2 \xrightarrow{\texttt{xs}} 1,\ 1 \xrightarrow{\texttt{input}} \mathbf{0},\ 10 \xrightarrow{\textbf{case}} 3,\ 10 \xrightarrow{\texttt{nil}} 4,\ 10 \xrightarrow{::(\texttt{y,ys})} 9,\ 3 \xrightarrow{\texttt{xs}} \mathbf{0},\ 4 \xrightarrow{\texttt{nil}} \mathbf{0},\ \ldots\}$$

Short form of the LTS for naive reverse (root state 2, `nr` *code start* `10`*, and* `ap` *code start* `17`*).* We abbreviate the LTS by omitting end-of-action state **0** and variable transitions to **0**, and bundling together transitions from a single state.

```
(2  -> (call 10 (input)))       ;  root = 2: call nr(input)
(10 -> (case xs ((nil).4) ((:: y ys).9))) ; start "nr"
(4  -> (nil))
```

```
(9  -> (call 17 (6 8))              ; call ap(nr(ys),...)
(6  -> (call 10 (ys)))              ; call nr(ys)
(8  -> (:: y 4))
(17 -> (case us ((nil).vs) ((:: u us1).16)))) ; start "ap"
(16 -> (:: u 15)
(15 -> (call 17 (us1 vs))           ; call ap(ws,vs)
```

An example of optimisation: The program above runs in time $O(n^2)$. It can, as is well known, be improved to run in time $O(n)$. Distillation does this automatically, yielding the following LTS with root state 3 and `rev` code start `10`. Effect: the nested loop in `nr` has been replaced by an accumulating parameter.

```
      ;  Reverse with an accumulating parameter
(3  -> (call 10 (us 2)))
(2  -> (nil))
(10 -> (case xs ((nil) . acc) ((:: x xs1) . 9)))
(9  -> (call 10 (xs1 8)))
(8  -> (:: x acc))
```

The distilled version in source language format.

```
rev us nil where

rev xs acc = case xs of
                 nil        => acc
               | (:: x xs1) => rev xs1 (:: x acc)
```

Are these programs bisimilar? There is no obvious bisimilarity relation between runtime states of `nr` and `rev`, e.g., because of different loop structures and numbers of variables. In the next section we will see that the *result of driving* a distilled program is always bisimilar to the *result of driving* the original program.

3 Distillation: a simplified version

We now describe (parts of) a cut-down version of distillation. Following the pattern of Turchin, Sørensen and Hamilton, the first step is driving.

3.1 Driving

Distillation and supercompilation of program $p = e$ **where** Δ both begin with an operation called *driving*. The result is an LTS $\mathcal{D}[\![p]\!]$, usually infinite, *with no function calls* and *with no free variables* other than those of p.

If p is closed, then driving will evaluate it completely, yielding as result an LTS for the value $[\![p]\!]$. Furthermore, given an LTS for a program p *with* free variables, the driver will:

- compute as much as can be seen to be computable;
- expand *all* function calls and
- yield as output a call-free LTS $\mathcal{D}[\![p]\!]$ equivalent to program p. (The output may be infinite if the input program has loops.)

$\mathcal{D}[\![p]\!]$ will consist entirely of constructors, variables and **case** expressions whose tests could not be resolved at driving time. This is a (usually infinite) LTS to compute the function $[\![p]\!]$ (of values of p's free variables). Another perspective: $\mathcal{D}[\![p]\!]$ is essentially a "glorified decision tree" to compute $[\![p]\!]$ without calls. Input is tested and decomposed by **case**, and output is built by constructors.

Although $\mathcal{D}[\![p]\!]$ may be infinite it is executable, given initial values of any free variables. This can be realised in a lazy language, where only a finite portion of the LTS is looked at in any terminating run.

Correctness of distillation: Theorem 3.10 in [11] shows that for any p, p',

$$\mathcal{D}[\![p]\!] \sim \mathcal{D}[\![p']\!] \textit{ implies } p \simeq p'$$

Bottom line: if two programs p, p' have bisimilar driven versions $\mathcal{D}[\![p]\!]$ and $\mathcal{D}[\![p']\!]$, then the programs are observationally equivalent.

3.2 A driver for the call-by-value language

The driving algorithm transforms a program into a call-free output LTS (possibly infinite). It is essentially an extended semantics: an expression evaluator that also allows free variables in the input (transitions to **0** are generated in the output LTS for these variables); and **case** edges that are applied to a non-constructor value (for each, a residual output LTS **case** transition is generated).

Relations to the drivers of [10–12]: We do not use silent transitions at all, and so do not need weak bisimulation. Our LTS states have no internal structure, i.e., they are not expressions as in [10–12], and have no syntactic information about the program from which they were generated, beyond function parameters and case pattern variables. (Embedding, generalisation, well-quasi-orders etc. are not discussed here, as this paper's points can be made without them.)

Another difference: the following constructs its output LTS "one state at a time": it explicitly allocates new states for constructors and for **case** expressions with unresolvable tests.[1]

One effect is an "instrumentation". For instance if p is closed, then the driven output LTS $\mathcal{D}[\![p]\!]$ will have one state for every constructor operation performed while computing $[\![p]\!]$, so $\mathcal{D}[\![\_]\!]$ yields *some intensional information* about its program argument's running time (in spite of Theorem 3.10 0f [11]).

Our language is call-by-value, so environments map variables into states, rather than into expressions as in [10, 11]. Types used in the driver:

[1] To avoid non-termination of the program transformer itself, we assume the input does not contain nonproductive loops such as $f\ 0$ **where** $f\ x = f\ x$.

$\mathcal{D} : Expression \to LTS$
$\mathcal{D}' : Expression \to LTS \to Environment \to FcnEnv \to LTS$
$\theta \in Environment = Variable \rightharpoonup State$
$\Delta \in FcnEnv = FunctionName \rightharpoonup Variable^* \rightharpoonup Expression$

Variable t ranges over LTS's, and s ranges over states. For brevity, function environment argument Δ in the definition of $\mathcal{D}'$ is elided since it is never changed.

1. $\mathcal{D}[\![e \textbf{ where } \Delta]\!] = \mathcal{D}'[\![e]\!]\ \varnothing\ \{\}\ \Delta$
2. $\mathcal{D}'[\![x]\!]\ t\ \theta\ = \begin{cases} t \text{ with root } \theta x & \textbf{if } x \in dom(\theta) \textbf{ else} \\ t \cup \{s \xrightarrow{x} \mathbf{0}\} & \textbf{where } s \text{ is a new root state} \end{cases}$
3. $\mathcal{D}'[\![c\ e_1 \ldots e_k]\!]\ t_0\ \theta\ =$ **let** $t_1 = \mathcal{D}'[\![e_1]\!]\ t_0\ \theta, \ldots, t_k = \mathcal{D}'[\![e_k]\!]\ t_{k-1}\ \theta$ **in**
   $t_k \cup \{s \xrightarrow{c} \mathbf{0}, s \xrightarrow{\#1} root(t_1), \ldots, s \xrightarrow{\#k} root(t_k)\}$ **where** s is a new root state
4. $\mathcal{D}'[\![f\ e_1 \ldots e_n]\!]\ t_0\ \theta\ =$ **let** $t_1 = \mathcal{D}'[\![e_1]\!]\ t_0\ \theta, \ldots, t_k = \mathcal{D}'[\![e_k]\!]\ t_{k-1}\ \theta$ **in**
   $\mathcal{D}'[\![e^f]\!]\ t_n\ \{x_1 \mapsto root(t_1), \ldots, x_n \mapsto root(t_n)\}$ **where** $\Delta\ f\ x_1 \ldots x_n = e^f$
5. $\mathcal{D}'[\![\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 | \ldots | p_n \Rightarrow e_n]\!]\ t\ \theta\ =$ **let** $t_0 = \mathcal{D}'[\![e_0]\!]\ t\ \theta$ **in**
   if $t_0 \ni s_0 \xrightarrow{c} \mathbf{0}, s_0 \xrightarrow{\#1} s_1, \ldots, s_0 \xrightarrow{\#k} s_k$ **and** $p_i = c\ x_1 \ldots x_k$
   then
   $\mathcal{D}'[\![e_i]\!]\ t_0\ (\theta \cup \{x_1 \mapsto s_1, \ldots, x_k \mapsto s_k\})$
   else
   let $t_1 = \mathcal{D}'[\![e_1]\!]\ t_0\ \theta, \ldots, t_n = \mathcal{D}'[\![e_n]\!]\ t_{n-1}\ \theta$ **in**
   $t_n \cup \{s \xrightarrow{\textbf{case}} root(t_0), s \xrightarrow{p_1} root(t_1), \ldots, s \xrightarrow{p_n} root(t_n)\}$
   where s is a new root state

3.3 Distillation's sequence of transformations

As presented in [11, 12], further analyses and transformations (homeomorphic embedding, generalisation, folding, etc.) on an infinite $\mathcal{D}[\![p]\!]$ will yield a finite transformed program p'. Furthermore, these transformations preserve the property of bisimilarity with $\mathcal{D}[\![p]\!]$.

The following may help visualise the various stages involved in distillation:

p		LTS^{in}		LTS^{driven}		LTS^{out}		p'
source program	**[parse]**	(finite, with calls)	**[drive]**	(infinite, no calls)	**[distill]**	(finite, with calls)	**[unparse]**	transformed program

Function $\mathcal{D}$ is **[drive]** $\circ$ **[parse]**.

4 Some speedup principles and examples

4.1 On sources of speedup by distillation

Speedups can be obtained for all our well-known "old chestnut" programs p as follows (where p_s is p applied to known values s of its free variables):

1. Drive: construct $\text{LTS}^{driven} = \mathcal{D}[\![p_s]\!]$ from p_s. (This is finite if p_s terminates.)
2. Remove "dead code" from LTS^{driven}: these are any states that are unreachable from its root.
3. Merge any bisimilar states in LTS^{driven}.

Step 2 must be done after constructing LTS^{driven}. Step 3 can be done either after or during driving: elide adding a new state and transitions $s \xrightarrow{a_1} s_1, \ldots, s \xrightarrow{a_n} s_n$ to LTS^{driven} if an already-existing LTS state has the same transitions.

Two points: first, in traditional compiler construction, dead code elimination is very familiar; whereas merging bisimilar states is a form of code folding not often seen (exception: the "rewinding" by Debois [6]). Distillation accomplishes the effect of both optimisations, and in some cases more sophisticated transformations.

Second, the distiller obtains superlinear speedup for all three programs by introducing *accumulating parameters*. In some cases, e.g., Fibonacci, the speedup is comparable to that of "tupling" of Chin et. al. [4,5]; but distillation does not introduce new constructors.

4.2 Overview of the "old chestnut" examples

Our goal is to relate the efficiencies of a program p and its distilled version p'. The transformation sequence as in Section 3.3 involves the possibly infinite object $\mathcal{D}[\![p]\!] = \text{LTS}^{driven}$.

The following experimental results get around this problem by computing $\mathcal{D}[\![p_s]\!]$ for *fixed* input values s. The idea is to drive a version p_s of p applied to known values s of its free variables. Assuming that p_s terminates, this will yield a finite LTS whose structure can be examined.

Let n be the input size (e.g., a list length or number value). Then

1. The naive reverse algorithm *nrev* runs in quadratic time, while its distilled version runs in linear time. Nonetheless, their driven versions are (strongly) bisimilar, and so observationally equivalent.
   Explanation of speedup: $\mathcal{D}[\![nrev_{(a_1 a_2 \ldots a_n)}]\!]$ has $O(n^2)$ states, including states for the computation of the reverse of every suffix of $(a_1 a_2 \ldots a_n)$. Among these, at the end of execution only $O(n)$ states are live, for the reverse of the full list $(a_1 a_2 \ldots a_n)$.
2. The naive program to compute Factorial sum ($sumfac(n) = 0! + 1! + \ldots n!$) has running time $O(n^2)$ and allocates $O(n^2)$ heap cells, due to repeated recomputation of $0!, 1!, 2!, \ldots$; but the distilled version is linear-time. The two are (again) observationally equivalent since their driven versions are bisimilar. The driven naive Factorial sum LTS has $O(n^2)$ states, but among these, only $O(n)$ are live at the end of execution.
   This example is interesting because both source and transformed programs are purely *tail recursive*, and so typical of compiler intermediate code.
3. A more extreme example: the obvious program fib for the Fibonacci function takes exponential time and will fill up the heap with exponentially many

memory cells. On the other hand, the distilled version of Fibonacci uses an accumulator and runs in linear time (counting $+, *$ as constant-time operations). Even so, the two LTS's are bisimilar.

In contrast to the examples above, the driven program $\mathcal{D}[\![fib_n]\!]$ has $O(1.7^n)$ states, all of which are live. Here speedup source number 3 (Section 4.1) comes into play: only $O(n)$ states are bisimulation-nonequivalent.

The experiments were carried out in SCHEME. The first step was parsing: to transform the input program from the form of Section 2.1 into an LTS, which for clarity we will call LTS^{in}. The driver as implemented realises the one of Section 3.2 (except that it it works on LTS^{in} rather than program p). LTS^{out} is the name of the distiller's output.

5 Can distillation save time by using space?

5.1 Palindrome: an experiment with an unexpected outcome

Long-standing open questions in complexity theory concern the extent to which computation time can be traded off against computation space. Consider the set

$$Pal = \{a_1 a_2 \ldots a_n \mid 1 \leqslant i \leqslant n \Rightarrow a_i = a_{n+1-i} \in \{0, 1\}\}$$

This set in LOGSPACE is decidable by a two-loop `cons`-free program that runs in time $O(n^2)$.

On the other hand, it can also be decided *in linear time* by a simple program *with* `cons`. The idea is first to compute the reverse of the input $xs = a_1 a_2 \ldots a_n$ by using an accumulating parameter; and then to compare xs to its reverse. Both steps can be done in linear time.

Here, using extra storage space (`cons`) led to reduced computation time.

A natural conjecture was that *any* `cons`*-free program deciding membership in Pal must run in superlinear time.* The reasoning was that one would not expect distillation to transform a `cons`-free program into one containing `cons`, as this would involve inventing a constructor not present in the input program. To test this conjecture, we ran an existing distiller on the `cons`-free program palindrome-decider.

The result was unexpected, and disproved our conjecture: distillation yielded a *linear-time* but second-order palindrome recogniser(!)

In effect, the distillation output realises `cons` by means of *second-order functions*. Thus, while it does not create any new `cons`'s its output program, it achieves a similar effect through the use of lambdas. The output is as follows[2]:

[2] Automatically produced but postprocessed by hand to increase readability.

```
p xs xs (λzs.True) where

p xs ys q = case xs of
                Nil       => q ys
              | (:: u us) => p us ys (r q u)

r t u = λws.case ws of
                Nil       => True
              | (:: v vs) => case u of
                                  0 => (case v of
                                            0 => t vs
                                          | 1 => False)
                               |  1 => (case v of
                                            0 => False
                                          | 1 => t vs)
```

Furthermore, this `cons`-free second-order program is tail recursive in the sense of Definition 6.13 from [15][3]:

Definition 6. `Cons`*-free program* `p` *is* higher-order tail recursive *if there is a partial order* $\geqslant$ *on function names such that any application* $\mathtt{f}\ \mathtt{x}_1 \ldots \mathtt{x}_m = \ldots \mathtt{e}_1\ \mathtt{e}_2 \ldots$ *such that* $\mathtt{e}_1$ *can evaluate to a closure* $\langle \mathtt{g}, v_1 \ldots v_{arity(\mathtt{g})-1} \rangle$ *satisfies either: (a)* $\mathtt{f} > \mathtt{g}$*, or (b)* $\mathtt{f} \equiv \mathtt{g}$ *and the call* $(\mathtt{e}_1\ \mathtt{e}_2)$ *is in tail position.*

The partial order `p`$>$`r` suffices for the palindrome program.

5.2 A theorem and another conjecture

How general is it that distillation sometimes achieves asymptotic speedups? Are the speedups observed in Palindrome, Naive reverse, Factorial Sum and Fibonacci function accidental? Is there a wide class of programs that the distiller can speed up significantly, e.g., from exponential time to polynomial time?

A lead: Jones [15] studies the computational complexity of `cons`-free programs. Two results from [15] about `cons`-free programs of type `[Bool] -> Bool` in our language: Given a set L of finite bit strings:

1. $L \in$ LOGSPACE iff L is decidable by a first-order `cons`-free program that is tail-recursive
2. $L \in$ PTIME iff L is decidable by a first-order `cons`-free program (not necessarily tail-recursive)

Beauty flaw: The result concerning PTIME, while elegant in form, is tantalising because the very `cons`-free programs that decide exactly problems in PTIME, in general *run in exponential time.* (This was pointed out in [15].)

This problem's source is repeated recomputation: subproblems are solved again and again. A tiny example with exponential running time:

[3] The definition is semantic, referring to all program executions, and so undecidable in general. Abstract interpretation can, however, safely approximate it.

```
f x = if x = [] then True else
      if f(tl x) then f(tl x) else False
```

This can be trivially optimised to linear time, but more subtle examples exist. More generally, Figure 5 in [15] builds from any PTIME Turing machine Z a first-order `cons`-free program `p` that simulates Z. In general `p` solves many subproblems repeatedly; these are not easily optimised away as in the tiny example.

The reason, intuitively: absence of `cons` means that earlier-computed results must be recomputed when needed, as they cannot be retrieved from a store.

The "trick" the distiller used in the Palindrome example was to speed up the given `cons`-free program (first-order, quadratic-time) by adding a function as an argument. The resulting second-order Palindrome program ran in linear time.

We now generalise, showing that *for any* first-order `cons`-free program, even one running in exponential time, there is a polynomial-time equivalent.[4]

Theorem 1. *L is decidable by a first-order* `cons`*-free program iff L is decidable by a second-order* `cons`*-free program* that runs in polynomial time.

Corollary 1. *$L \in$ PTIME iff L is decidable by a second-order* `cons`*-free program that runs in polynomial time.*

Some comments before sketching the proof. First, the condition "that runs in polynomial time," while wordy, is necessary since (as shown in [15]), unrestricted second-order cons-free programs decide exactly the class EXPTIME, a proper superset of PTIME. In fact, second-order cons-free programs can run in double exponential time (another variation on the "beauty flaw" remarks above).

Second, the Corollary is analogous to the standard definition: $L \in$ PTIME iff it is decidable by a Turing machine that runs in polynomial time. The punch line is that no tape or other form of explicit storage is needed; it is enough to allow functions as arguments.

Proof (sketch "if"). Suppose L is decidable by a second-order `cons`-free program that runs in polynomial time. All that is needed is to ensure that L can also be decided by a polynomial-time Turing machine. This follows by the known time behavior of call-by-value implementations of functional programs, e.g., as done by traditional compilers that represent functional values by closures.

Proof (sketch "only if"). First, suppose first-order `cons`-free program `p` decides L. Consider the "cache-based algorithm" to compute $[\![\texttt{p}]\!]\,\texttt{as}$ shown in Figure 8 of [15]. While this runs in polynomial time by *Theorem 7.16*, it is not `cons`-free since it uses storage for the cache.

Second, Figure 8 can be reprogrammed, to replace the cache argument by a second-order function. Rather than give a general construction, we just illustrate the idea for the Fibonacci function, and leave the reader to formulate the general construction. The standard definition of Fibonacci:

[4] Can this be strengthened to linear time? No, since $time(O(n)) \subsetneq time(O(n^2))$.

```
f n = if n <= 1 then 1 else f(n-1) + f(n-2)
```

The "cache" of Figure 8 in [15] is a table. For each function f in `p`, it contains all the arguments and the results $f(arguments)$ that have been computed so far. For a cons-free first-order program and an input of length n there can only be polynomially many different arguments.

Of course the cache of [15] requires some form of storage, e.g., `cons`. The trick in this paper is to go to second-order cons-free form by replacing the cache by a function $c : C = (Arguments \to Outputs)$ and simply applying c when arguments are to be looked up in the cache. The cache c must be updated at the return of every function, so $f : A \to B$ is replaced by $f' : A \times C \to B \times C$.

A cached version for the concrete case of the Fibonacci function is:

```
f n (λn.0) where
f n c =
let cv = c n in
  if cv /= 0 then (cv,c) else
    if n <= 1 then (1,update c n 1) else
      let (u,c1) = f (n-1) c in
        let (v, c2) = f (n-2) c1 in
          let r = u+v in (r,update c n r)
update c n v =
   if c n == v then c else λm.if m==n then v else c m
```

This program runs in polynomial time: it makes $O(n)$ calls of `f` when computing `f(n)`, and the time spent checking the cache contents is also polynomially limited. ⊣

A conjecture strengthening Theorem 1:

Distillation can transform any first-order `cons`-free program into an equivalent second-order `cons`-free program that runs in polynomial time.

Basis for the conjecture: it seems plausible that distillation, if applied to an arbitrary first-order `cons`-free program `p`, can transform it into an equivalent second-order program that runs in polynomial time. Reasoning: the transformation of the proof above, if applied to a general first-order `cons`-free program, seems analogous to the transformation that the distiller realised for the "palindrome" program.

More generally, each of the "accumulating parameters" that distillation generated for the reverse, factorial sum and Fibonacci examples resembles a cache with static structure. Renaining to investigate is whether distillation can yield an accumulating parameter the corresponds to a cache with dynamic structure, as seen in the previous program. (Perhaps a static analysis of the Fiboonacci code above could reveal that `c` is used in a static manner.)

6 Final remarks and conclusions

6.1 A question to be resolved

A better understanding is emerging on the source of these interesting program optimisations, though some questions are still less than perfectly clear. An example: although the "supercompilation" that distillation is based on [12, 21, 23] yields at most linear speedups, distillation sometimes achieves superlinear speedups. The major technical difference (at transformation-time) is that distillation does "generalisation" by a form of second-order pattern matching.

The question: why and how does this make such a difference in the efficiency of transformed programs? Answering this will require a better global insight into second-order generalisation.

6.2 Are there limits to speedup by distillation?

The fact that distillation often yields linear-time programs may at first seem to conflict with well-known results from complexity theory [14]: for example, for any computable function f, there exist computational problems that cannot be solved in time $\leqslant f(n)$ by any program. Consequence: there must be some limit to how much transformation techniques such as distillation can achieve, regardless of how strong the techniques used are.

An interesting question: Is there some sense in which distillation achieves a best possible result, e.g., analogous to a minimal-state finite automaton? This might be so.

However, by Blum's speedup theorem [2] some functions have no best program, precluding the possibility that distillation can always achieve the best possible result in terms of efficiency. Furthermore, the output of distillation must always be a finite program. This requirement could force the output program to be asymptotically less efficient than an infinite LTS $\mathcal{D}[\![p]\!]$ resulting from driving the input program.

6.3 An analogy with the Myhill-Nerode theorem

The Myhill-Nerode theorem [20] concerns definability of sets of finite strings over a finite alphabet, for example a set $L \subseteq \{0,1\}^*$. The starting point is to define an equivalence relation over finite strings $x, y \in \{0,1\}^*$ by

$$x \equiv y \text{ iff } \forall z \;.\; (xz \in L \Leftrightarrow yz \in L)$$

Theorem L is a regular set if and only if the relation $\equiv$ has only finitely many equivalence classes. Furthermore, a minimal-state finite automaton M_L that accepts exactly L can be constructed from $\equiv$.

An interesting fact: the relation $\equiv$ is well-defined for *any* subset $L \subseteq \{0,1\}^*$, whether regular or not. If L is not regular, then M_L will have infinitely many states. In all cases, M_L is a homomorphic image of any automaton (finite- or infinite-state) that accepts L.

A consequence is that one can perform *state minimisation* of an initial automaton M by first constructing the relation $\equiv$ for the set accepted by M, and then constructing M_L from the equivalence classes of $\equiv$.

The analogy: In the case of distillation, an initial program p is given, and the possibly infinite LTS $\mathcal{D}[\![p]\!]$ is constructed from it by driving. Once this is available, the distillation step is applied to construct from it another (finite) program p' that will often be faster than the original program p.

While the goal criteria for theMyhill-Nerode construction and distillation differ (smaller-size state sets for DFA minimisation versus asymptotically faster programs for distillation), the overall pattern seems tantalisingly similar.

6.4 Conclusions

In spite of many remaining open quesions, we hope the material above, particularly Sections 3 and 4, clarifies the way that distillation can yield superlinear program speedups.

The question "how can an $O(n^2)$ program or $O(2^n)$ program be bisimilar to an $O(n)$ program?" has been answered: It is not the runtime state transitions of the two programs that are bisimilar; but rather their driven versions. Furthermore, the numbers of states in their driven versions trace the number of `cons`'s performed, and so reflect the two programs' relative running times.

Finally, a large program set has been identified in which superlinear speedups are likely to be achievable by by distillation: the first-order `cons`-free programs.

Acknowledgement: This paper has been much improved as a result of discussions with Luke Ong and Jonathan Kochems at Oxford University. Referee comments, particularly by Neil Mitchell, were very useful. The work was supported, in part, by DIKU at the University of Copenhagen, and by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

1. Bird, R.: Improving programs by the introduction of recursion. Commun. ACM 20(11), 856–863 (1977)
2. Blum, M.: A machine independent theory of the complexity of recursive functions. J. ACM 14, 322–336 (1967)
3. Burstall, R., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (Jan 1977)
4. Chin, W.N.: Towards an automated tupling strategy. In: PEPM. pp. 119–132. ACM (1993)
5. Chin, W.N., Khoo, S.C., Jones, N.: Redundant call elimination via tupling. Fundam. Inform. 69(1-2), 1–37 (2006)
6. Debois, S.: Imperative program optimization by partial evaluation. In: PEPM (ACM SIGPLAN 2004 Workshop on Partial Evaluation and Program Manipulation). pp. 113–122 (2004)

7. Ershov, A.P.: On the essence of compilation. In: Neuhold, E. (ed.) Formal Description of Programming Concepts. pp. 391–420. MAsterdam: North-Holland (1978)
8. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12(4), 381–391 (1999)
9. Gordon, A.D.: Bisimilarity as a theory of functional programming. Theoretical Computer Science 228(1–2), 5–47 (1999)
10. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 61–70 (2007)
11. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems. In: PEPM (ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation). pp. 15–24. ACM (2012)
12. Hamilton, G.W., Jones, N.D.: Proving the correctness of unfold/fold program transformations using bisimulation. In: Proceedings of the 8th Andrei Ershov Informatics Conference. Lecture Notes in Computer Science, vol. 7162, pp. 150–166. Springer (2012)
13. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
14. Jones, N.D.: Computability and complexity - from a programming perspective. Foundations of computing series, MIT Press (1997)
15. Jones, N.D.: The expressive power of higher-order types or, life without cons. J. Funct. Program. 11(1), 5–94 (2001)
16. Jones, N.D.: Transformation by interpreter specialisation. Sci. Comput. Program. 52, 307–339 (2004)
17. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Compiler optimization correctness by temporal logic. Higher-Order and Symbolic Computation 17(3), 173–206 (2004)
18. Lerner, S., Millstein, T.D., Chambers, C.: Cobalt: A language for writing provably-sound compiler optimizations. Electr. Notes Theor. Comput. Sci. 132(1), 5–17 (2005)
19. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
20. Nerode, A.: Linear automaton transformations. In: Proceedings of the AMS 9. pp. 15–24. AMS (1958)
21. Sørensen, M.H., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
22. Turchin, V.F.: Supercompilation: Techniques and results. In: Perspectives of System Informatics. LNCS, vol. 1181. Springer (1996)
23. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 90–121 (Jul 1986)
24. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: Ganzinger, H. (ed.) ESOP'88. 2nd European Symposium on Programming, Nancy, France, March 1988 (Lecture Notes in Computer Science, vol. 300). pp. 344–358 (1988)