Distilling Programs to Prove Termination

G.W. Hamilton

School of Computing
Dublin City University
Ireland

`hamilton@computing.dcu.ie`

The problem of determining whether or not any program terminates was shown to be undecidable by Turing, but recent advances in the area have allowed this information to be determined for a large class of programs. The classic method for deciding whether a program terminates dates back to Turing himself and involves finding a *ranking function* that maps a program state to a well-order, and then proving that the result of this function decreases for every possible program transition. More recent approaches to proving termination have involved moving away from the search for a single ranking function and toward a search for a set of ranking functions; this set is a choice of ranking functions and a disjunctive termination argument is used. In this paper, we describe a new technique for determining whether programs terminate. Our technique is applied to the output of the *distillation* program transformation that converts programs into a simplified form called *distilled form*. Programs in distilled form are converted into a corresponding *labelled transition system* and termination can be demonstrated by showing that all possible infinite *traces* through this labelled transition system would result in an infinite descent of well-founded data values. We demonstrate our technique on a number of examples, and compare it to previous work.

1 Introduction

The *program termination problem*, or *halting problem*, can be defined as follows: using only a finite amount of time, determine whether a given program will always finish running or could execute forever. This problem rose to prominence before the development of stored program computers, in the time of Hilbert's *Entscheidungs problem*: the challenge to formalise all of mathematics and use algorithmic means to determine the validity of all statements. The halting problem was famously shown to be undecidable by Turing [26].

Although it is not possible to prove program termination in all cases, there are many programs for which this can be proved. The classic method for doing this dates back to Turing himself [27] and involves finding a ranking function that maps a program state to a well-order, and then proving that the result of this function decreases for every possible program transition. This has a number of useful applications, such as in program verification, where *partial correctness* is often proved using deductive methods and a separate proof of termination is given to show *total correctness*, as originally done by Floyd [11]. More recent approaches to proving termination have involved moving away from the search for a single ranking function and toward a search for a set of ranking functions; this set is a choice of ranking functions and a disjunctive termination argument is used. Program termination techniques have been developed for functional programs [12, 17, 14, 19], logic programs [20, 7, 18], term rewriting systems [10, 2, 24] and imperative programs [11, 5, 3, 8, 1, 23, 9, 13].

In this paper, we describe a new approach to the termination analysis of functional programs that is applied to the output of the *distillation* program transformation [15, 16]. Distillation converts programs into a simplified form called *distilled form*, and to prove that programs in this form terminate, we convert them into a corresponding *labelled transition system* and then show that all possible infinite *traces*

through the labelled transition system would result in an infinite descent of well-founded data values. This proof of termination is similar to that described in [6] using *cyclic proof* techniques. However, we are able to prove termination for a wider class of programs.

The language used throughout this paper is a call-by-name higher-order functional language with the following syntax.

Definition 1.1 (Language Syntax) The syntax of this language is as shown in Fig. 1. $\qquad\square$

$$prog ::= e_0 \textbf{ where } h_1 = e_1, \ldots, h_k = e_k \qquad \text{Program}$$

e	::=	x	Variable
	|	$c\ e_1 \ldots e_n$	Constructor Application
	|	$\lambda x.e$	λ-Abstraction
	|	f	Function Call
	|	$e_0\ e_1$	Application
	|	$\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$	Case Expression
	|	$\textbf{let } x = e_0 \textbf{ in } e_1$	Let Expression
h	::=	$f\ x_1 \ldots x_n$	Function Header
p	::=	$c\ x_1 \ldots x_n$	Pattern

Figure 1: Language Syntax

Programs in the language consist of an expression to evaluate and a set of function definitions. An expression can be a variable, constructor application, λ-abstraction, function call, application, **case** or **let**. Variables introduced by function headers, λ-abstractions, **case** patterns and **let**s are *bound*; all other variables are *free*. An expression that contains no free variables is said to be *closed*. We write $e \equiv e'$ if e and e' differ only in the names of bound variables.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. It is also assumed that erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x.e)$ **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$ cannot occur.

Example 1 Consider the program from [4] shown in Figure 2 for calculating the greatest common divisor of two numbers x and y. Proving the termination of this program is tricky as there is no clear continued decrease in the size of either of the parameters of the *gcd* function (even though a number is subtracted from one of the arguments in each recursive call, it is difficult to determine that the number subtracted must be non-zero). We show how the termination of this program can be proved using our approach.

The remainder of this paper is structured as follows. In Section 2, we give some preliminary definitions that are used throughout the paper. In Section 3, we define the *labelled transition systems* that are used in our termination proofs. In Section 4, we show how to prove termination of programs using our technique, and apply this technique to the program in Figure 2. In Section 5, we give some examples of programs that cause difficulties in termination analysis using other techniques, but are shown to terminate using our technique. Section 6 concludes and considers related work.

$$gcd\ x\ y$$
where
$$gcd\ x\ y = \textbf{case}\ (gt\ x\ y)\ \textbf{of}$$
$$True\ \rightarrow gcd\ (sub\ x\ y)\ y$$
$$\mid False \rightarrow \textbf{case}\ (gt\ y\ x)\ \textbf{of}$$
$$True\ \rightarrow gcd\ x\ (sub\ y\ x)$$
$$\mid False \rightarrow x$$
$$gt\ x\ y\ \ = \textbf{case}\ x\ \textbf{of}$$
$$Zero\ \ \rightarrow False$$
$$\mid Succ\ x \rightarrow \textbf{case}\ y\ \textbf{of}$$
$$Zero\ \ \rightarrow True$$
$$\mid Succ\ y \rightarrow gt\ x\ y$$
$$sub\ x\ y = \textbf{case}\ y\ \textbf{of}$$
$$Zero\ \ \rightarrow x$$
$$\mid Succ\ y \rightarrow \textbf{case}\ x\ \textbf{of}$$
$$Zero\ \ \rightarrow Zero$$
$$\mid Succ\ x \rightarrow sub\ x\ y$$

Figure 2: Example Program

2 Preliminaries

In this section, we complete the presentation of our programming language and give a brief overview of the distillation program transformation algorithm.

2.1 Language Definition

Definition 2.1 (Substitution) We use the notation $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ to denote a *substitution*. If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.   □

Definition 2.2 (Language Semantics) The call-by-name operational semantics of our language is standard: we define an evaluation relation $\Downarrow$ between closed expressions and *values*, where values are expressions in *weak head normal form* (i.e. constructor applications or λ-abstractions). We define a one-step reduction relation $\overset{r}{\rightsquigarrow}$ inductively as shown in Fig. 3, where the reduction r can be f (unfolding of function f), c (elimination of constructor c) or β (β-substitution).   □

Definition 2.3 (Context) A context C is an expression with a "hole" [] in the place of one sub-expression. $C[e]$ is the expression obtained by replacing the hole in context C with the expression e. The free variables within e may become bound within $C[e]$; if $C[e]$ is closed then we call it a *closing context* for e.

We use the notation $e \overset{r}{\rightsquigarrow}$ if the expression e reduces, $e \Uparrow$ if e diverges, $e \Downarrow$ if e converges and $e \Downarrow v$ if e evaluates to the value v. These are defined as follows, where $\overset{r}{\rightsquigarrow}^*$ denotes the reflexive transitive closure of $\overset{r}{\rightsquigarrow}$:

$$((\lambda x.e_0)\ e_1) \overset{\beta}{\leadsto} (e_0\{x \mapsto e_1\}) \qquad \frac{f\ x_1 \ldots x_n = e}{f \overset{f}{\leadsto} \lambda x_1 \ldots x_n.e} \qquad \frac{e_0 \overset{r}{\leadsto} e_0'}{(e_0\ e_1) \overset{r}{\leadsto} (e_0'\ e_1)}$$

$$\frac{p_i = c\ x_1 \ldots x_n}{(\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e_1'|\ldots|p_k : e_k') \overset{c}{\leadsto} (e_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})}$$

$$\frac{e_0 \overset{r}{\leadsto} e_0'}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1|\ldots p_k : e_k) \overset{r}{\leadsto} (\textbf{case}\ e_0'\ \textbf{of}\ p_1 : e_1|\ldots p_k : e_k)}$$

$$(\textbf{let}\ x = e_0\ \textbf{in}\ e_1) \overset{\beta}{\leadsto} (e_1\{x \mapsto e_0\})$$

Figure 3: One-Step Reduction Relation

$$e \overset{r}{\leadsto}, \text{ iff } \exists e'.e \overset{r}{\leadsto} e' \qquad\qquad e \Downarrow, \text{ iff } \exists v.e \Downarrow v$$
$$e \Downarrow v, \text{ iff } e \overset{r}{\leadsto}^* v \wedge \neg(v \overset{r}{\leadsto}) \qquad\qquad e \Uparrow, \text{ iff } \forall e'.e \overset{r}{\leadsto}^* e' \Rightarrow e' \overset{r}{\leadsto}$$

Definition 2.4 (Contextual Equivalence) Contextual equivalence, denoted by $\simeq$, equates two expressions if and only if they exhibit the same termination behaviour in all closing contexts i.e. $e_1 \simeq e_2$ iff $\forall C . C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$.

2.2 Distillation

Distillation [15, 16] is a powerful program transformation technique that builds on top of the positive supercompilation transformation algorithm [25, 22]. The following theorems have previously been proved about the distillation transformation $\mathscr{D}$.

Theorem 2.5 (Correctness of Transformation) $\forall p \in Prog : \mathscr{D}[\![p]\!] \simeq p$

Thus, the resulting program will have the same termination properties as the original program in all contexts.

Theorem 2.6 (On The Form of Expressions Produced by Distillation) For all possible input programs, distillation terminates and the form of expressions it produces (after all function arguments that are not variables are extracted using **let**s), which we call *distilled form*, is described by $e^{\emptyset}$ where e^{ρ} is defined as follows:

$$
\begin{aligned}
e^{\rho} \quad ::= \quad & x\ e_1^{\rho} \ldots e_n^{\rho} \\
| \quad & c\ e_1^{\rho} \ldots e_n^{\rho} \\
| \quad & \lambda x.e^{\rho} \\
| \quad & f\ x_1 \ldots x_n\ (\text{where } f \text{ is defined by } f\ x_1 \ldots x_n = e^{\rho}) \\
| \quad & \textbf{case}\ (x\ e_1^{\rho} \ldots e_n^{\rho})\ \textbf{of}\ p_1 \Rightarrow e_{n+1}^{\rho} |\cdots| p_k \Rightarrow e_{n+k}^{\rho}\ (x \notin \rho) \\
| \quad & \textbf{let}\ x = e_0^{\rho}\ \textbf{in}\ e_1^{(\rho \cup \{x\})}
\end{aligned}
$$

The particular property of expressions in distilled form that makes them easier to analyse for termination is that no sub-expression that has been extracted using a **let** expression can be an intermediate data structure; **let** variables are added to the set ρ, and cannot be used in the selectors of **case** expressions. This means that once a parameter has increased in size it cannot subsequently decrease, which makes it much easier to identify parameters that decrease in size.

Due to space considerations, we are not able to include a full definition of the distillation algorithm here. However, we can simply treat this as a black box that does not alter the termination properties of a program and will always convert it into distilled form, so this paper is still reasonably self-contained.

Example 2 The result of transforming the example program in Figure 2 is shown in Figure 4. We can see that this program is indeed in distilled form.

$$
\begin{array}{l}
\textit{f0 x y x y} \\
\textbf{where} \\
\textit{f0 a b c d} = \textbf{case } a \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow c \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f1 c b c b} \\
\qquad\qquad\qquad |\ \textit{Succ a} \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{f3 a d a d} \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f0 a b c d} \\
\textit{f1 a b c d} = \textbf{case } a \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{f0 c b c b} \\
\qquad\qquad\qquad |\ \textit{Succ a} \rightarrow \textit{f2 a b c d} \\
\textit{f2 a b c d} = \textbf{case } a \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow c \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f1 c b c b} \\
\qquad\qquad\qquad |\ \textit{Succ a} \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{f5 a d a d} \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f2 a b c d} \\
\textit{f3 a b c d} = \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{f3 a d a d} \\
\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f4 a b c d} \\
\textit{f4 a b c d} = \textbf{case } a \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{Succ c} \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f5 c b c b} \\
\qquad\qquad\qquad |\ \textit{Succ a} \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{f3 a d a d} \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f4 a b c d} \\
\textit{f5 a b c d} = \textbf{case } a \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{Succ c} \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f5 c b c b} \\
\qquad\qquad\qquad |\ \textit{Succ a} \rightarrow \textbf{case } b \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad \rightarrow \textit{f5 a d a d} \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \textit{Succ b} \rightarrow \textit{f5 a b c d}
\end{array}
$$

Figure 4: Example Program Distilled

3 Labelled Transition Systems

In this section, we define the labelled transition systems used in our termination analysis.

Definition 3.1 (Labelled Transition System) A *labelled transition system* (LTS) is a 4-tuple $(\mathscr{E}, e_0, Act, \rightarrow)$ where:

- $\mathscr{E}$ is a set of *states* of the LTS. Each is an expression or the end-of-action state **0**.

- $e_0 \in \mathscr{E}$ is the *start state*.

- *Act* is a set of *actions* which can be one of the following:

    - x, a variable;
    - c, a constructor;
    - λx, a λ-abstraction;
    - f, a function unfolding;
    - @, the function in an application;
    - #i, the ith argument in an application;
    - **case**, a case selector;
    - p, a case pattern;
    - **let** x, a let variable
    - **in**, a let body.

- $\rightarrow \subseteq \mathscr{E} \times Act \times \mathscr{E}$ is a *transition relation*. We write $e \xrightarrow{\alpha} e'$ for a transition from state e to state e' via action α. □

We also write $e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)$ for a LTS with start state e where $t_1 \ldots t_n$ are the LTSs obtained by following the transitions labelled $\alpha_1 \ldots \alpha_n$ respectively from e.

Definition 3.2 (Renaming) We use the notation $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ to denote a *renaming*. If e is an expression, then $e\sigma = e\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ is the result of simultaneously replacing the free variables $x_1 \ldots x_n$ with the corresponding variables $x'_1 \ldots x'_n$ respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture. □

Definition 3.3 (Folded LTS) A *folded LTS* is a LTS which also contains renamings of the form $e \xrightarrow{\sigma} e'$, where σ is a renaming s.t. $e \equiv e'\sigma$. □

We now show how to generate the LTS representation of a program. It is assumed here that all function arguments in the program are variables; it is always possible to extract non-variable function arguments using **let**s to ensure that this is the case.

Definition 3.4 (Generating LTS From Program) A LTS can be generated from a program p as $\mathscr{L}_p[\![p]\!]$ using the rules as shown in Fig. 5. The rules $\mathscr{L}_e$ generate a LTS from an expression where the parameter ρ is the set of previously encountered function calls and the parameter Δ is the set of function definitions. If a renaming of a previously memoised function call is encountered, no further transitions are added to the constructed LTS. Thus, the constructed LTS will always be a finite representation of the program. □

Example 3 The LTS generated for the distilled *gcd* program in Figure 4 is shown in Figure 6.

$$\mathcal{L}_p[\![e_0 \textbf{ where } f_1 = e_1, \ldots, f_n = e_n]\!] = \mathcal{L}_e[\![e_0]\!] \; \rho \; (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})$$

$$
\begin{aligned}
&\mathcal{L}_e[\![e = x]\!] \; \rho \; \Delta && = e \to (x, \mathbf{0}) \\
&\mathcal{L}_e[\![e = c \; e_1 \ldots e_n]\!] \; \rho \; \Delta && = e \to (c, \mathbf{0}), (\#1, \mathcal{L}_e[\![e_1]\!] \; \rho \; \Delta), \ldots, (\#n, \mathcal{L}_e[\![e_n]\!] \; \rho \; \Delta) \\
&\mathcal{L}_e[\![e = \lambda x.e]\!] \; \rho \; \Delta && = e \to (\lambda x, \mathcal{L}_e[\![e]\!] \; \rho \; \Delta) \\
&\mathcal{L}_e[\![e = f \; x_1 \ldots x_n]\!] \; \rho \; \Delta && = \begin{cases} e \xrightarrow{\sigma} e', & \text{if } \exists e' \in \rho . e' \sigma \equiv e \\ e \to (f, \mathcal{L}_e[\![e']\!] \; (\rho \cup \{e\}) \; \Delta), & \text{otherwise } (f \equiv \lambda x_1 \ldots x_n.e' \in \Delta) \end{cases} \\
&\mathcal{L}_e[\![e = e_0 \; e_1]\!] \; \rho \; \Delta && = e \to (@, \mathcal{L}_e[\![e_0]\!] \; \rho \; \Delta), (\#1, \mathcal{L}_e[\![e_1]\!] \; \rho \; \Delta) \\
&\mathcal{L}_e[\![e = \textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n]\!] \; \rho \; \Delta && \\
& && = e \to (\textbf{case}, \mathcal{L}_e[\![e_0]\!] \; \rho \; \Delta), (p_1, \mathcal{L}_e[\![e_1]\!] \; \rho \; \Delta), \ldots, (p_n, \mathcal{L}_e[\![e_n]\!] \; \rho \; \Delta) \\
&\mathcal{L}_e[\![e = \textbf{let } x = e_0 \textbf{ in } e_1]\!] \; \rho \; \Delta && = e \to (\textbf{let } x, \mathcal{L}_e[\![e_0]\!] \; \rho \; \Delta), (\textbf{in}, \mathcal{L}_e[\![e_1]\!] \; \rho \; \Delta)
\end{aligned}
$$

Figure 5: LTS Representation of a Program

4 Proving Termination

In order to prove that a program terminates, we analyse the labelled transition system generated from the result of transforming the program using distillation. We need to show that within every cycle in this labelled transition system, at least one parameter is decreasing. We define a decreasing parameter as follows.

Definition 4.1 (Decreasing Parameter) A parameter is considered to decrease in size if it is the subject of a **case** selector.   □

A parameter that is the subject of a **case** selector is deconstructed into smaller components and therefore decreases in size. We define an increasing parameter as follows.

Definition 4.2 (Increasing Parameter) A parameter is considered to increase in size if any expression other than a variable is assigned to it.   □

Note that this is a conservative criterion for an increase in size based on the syntactic size of the parameter rather than the semantic size. Thus, for example, in the call *gcd (sub x y) y* , the first parameter would be considered to be increasing syntactically, even though it is actually decreasing semantically. However, such potentially increasing parameters are often transformed by distillation to reveal that they are in fact decreasing, as we have seen is the case for this example.

Lemma 4.3 (On Decreasing Parameters) Every parameter that has decreased in size cannot previously have increased in size.   □

Proof. This can be proved quite straightforwardly from the definition of distilled form in Theorem 2.6. Within the distilled form e^ρ, if any expression other than a variable is assigned to a parameter using a **let**, then the parameter is added to the set ρ and cannot subsequently be the subject of a **case** selector. Thus, if a parameter has increased in size, it cannot subsequently decrease.   □

In order to prove that a program terminates, we need to show that all possible *traces* through the labelled transition system generated from the result of distilling the program are *infinitely progressing*. We now define what these terms mean.

Definition 4.4 (Trace) A *trace* within a labelled transition system $(\mathcal{E}, e_0, Act, \to)$ is a sequence of states $e_0, e_1, \ldots$ where $\forall i. \exists \alpha. e_i \xrightarrow{\alpha} e_{i+1} \in \to$.   □

Figure 6: LTS Representation of Example Program

Definition 4.5 (Infinitely Progressing Trace) An *infinitely progressing trace* is a trace that contains an infinite number of decreases in parameter size. ☐

Theorem 4.6 (Termination) If all traces through the labelled transition system generated from the result of distilling a program are infinitely progressing, then the program terminates. ☐

Proof. From Lemma 4.3, every parameter that has decreased in size cannot previously have increased in size. If the trace is infinitely progressing, then there must be an infinite number of decreases in parameter size. As these parameters cannot have increased in size elsewhere within the trace, there must be infinite descent. ☐

Since a decreasing parameter must be the subject of a **case** selector, to show that a program terminates it is sufficient to show that in the labelled transition system generated from the result of distilling the program there is a **case** expression between every renamed state and its renaming.

Example 4 In the LTS generated from the distilled program in Figure 4 is shown in Figure 6, we can see that there is a **case** expression between every renamed state and its renaming, so this program is indeed terminating. Proving the termination of the original program is tricky as there is no clear continued decrease in the size of either of the parameters of the *gcd* function (even though a number is subtracted from one of the arguments in each recursive call, it is difficult to determine that the number subtracted must be non-zero).

5   Examples

We now give some examples of programs that cause difficulties in termination analysis using other techniques, but can be shown to terminate using the technique described here. None of these examples can be proven to terminate using the size-change principle described in [17].

Example 5 Consider the following program:

$$
\begin{aligned}
&f\ n \\
&\textbf{where} \\
&f\ n = \textbf{case}\ n\ \textbf{of} \\
&\qquad\quad Zero\quad \rightarrow Zero \\
&\qquad\quad |\ Succ\ n' \rightarrow g\ (Succ\ n) \\
&g\ n = \textbf{case}\ n\ \textbf{of} \\
&\qquad\quad Zero\quad \rightarrow Zero \\
&\qquad\quad |\ Succ\ n' \rightarrow \textbf{case}\ n'\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\quad Zero\quad \rightarrow Zero \\
&\qquad\qquad\qquad\qquad\quad |\ Succ\ n'' \rightarrow f\ n''
\end{aligned}
$$

This has mutually recursive functions *f* and *g*, where the parameter is increasing in the call from *f* to *g*, and decreasing in the call from *g* to *f* and therefore causes difficulties for other termination checkers. The result of transforming this program using distillation is as follows:

$$
\begin{aligned}
&f\ n \\
&\textbf{where} \\
&f\ n = \textbf{case}\ n\ \textbf{of} \\
&\qquad\quad Zero\quad \rightarrow Zero \\
&\qquad\quad |\ Succ\ n' \rightarrow f\ n'
\end{aligned}
$$

The LTS generated for this transformed program is shown in Figure 7. We can now quite easily see that this program terminates as there is a **case** expression between the function call $f\ n$ and its renaming $f\ n'$.
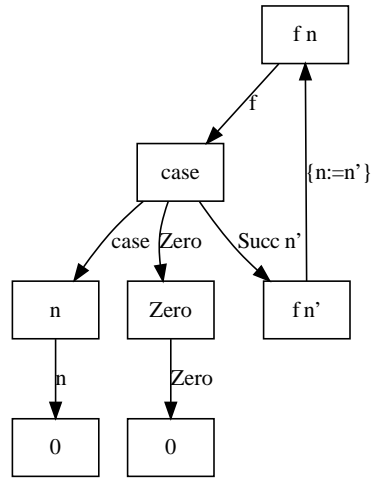

Figure 7: LTS Representation of Program in Example 5

Example 6 Consider the following program:

$$f\ m\ n$$
$$\textbf{where}$$
$$f\ m\ n\ =\ \textbf{case}\ m\ \textbf{of}$$
$$\qquad Zero\quad \rightarrow Zero$$
$$\qquad |\ Succ\ m' \rightarrow f\ (sub\ m\ n)\ (Succ\ n)$$
$$sub\ x\ y = \textbf{case}\ y\ \textbf{of}$$
$$\qquad Zero\quad \rightarrow x$$
$$\qquad |\ Succ\ y \rightarrow \textbf{case}\ x\ \textbf{of}$$
$$\qquad\qquad\qquad Zero\quad \rightarrow Zero$$
$$\qquad\qquad\qquad |\ Succ\ x \rightarrow sub\ x\ y$$

This causes problems for other termination checkers as the size of the second parameter is increasing and the size of the first parameter will not decrease if the value of the second parameter is *Zero*. The result of transforming this program using distillation is as follows:

$$f\ m\ n$$
$$\textbf{where}$$
$$f\ m\ n = \textbf{case}\ m\ \textbf{of}$$
$$\qquad Zero\quad \rightarrow Zero$$
$$\qquad |\ Succ\ m' \rightarrow \textbf{case}\ n\ \textbf{of}$$
$$\qquad\qquad\qquad Zero\quad \rightarrow g\ m'$$
$$\qquad\qquad\qquad |\ Succ\ n' \rightarrow f\ m'\ n'$$
$$g\ m\quad = \textbf{case}\ m\ \textbf{of}$$
$$\qquad Zero\quad \rightarrow Zero$$
$$\qquad |\ Succ\ m' \rightarrow g\ m'$$

The LTS generated for this transformed program is shown in Figure 8. We can see that there is a case expression between every renamed state and its renaming, so this program is indeed terminating.
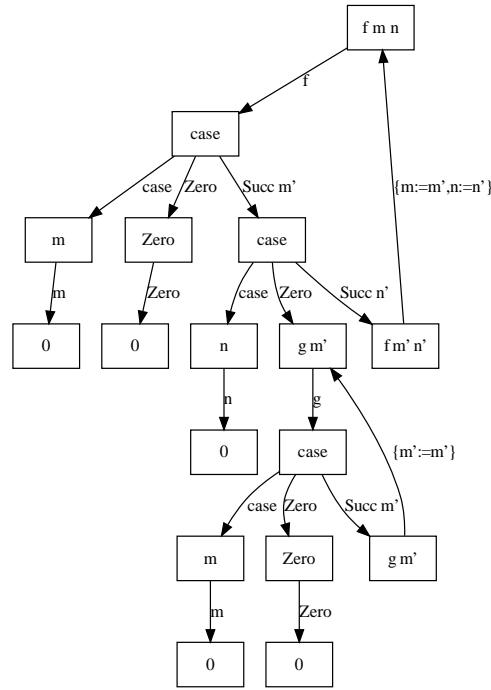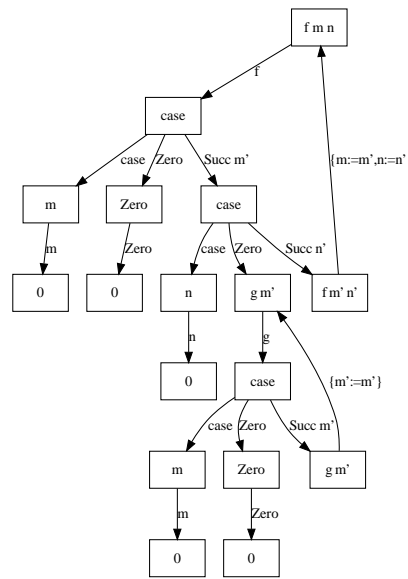

Figure 8: LTS Representation of Program in Example 5

Example 7 Consider the following program:

$$
\begin{aligned}
&f \ m \ n \\
&\textbf{where} \\
&f \ m \ n = \textbf{case } m \textbf{ of} \\
&\qquad\qquad Zero \quad\ \to Zero \\
&\qquad\quad\ | \ Succ \ m' \to \textbf{case } n \textbf{ of} \\
&\qquad\qquad\qquad\qquad\qquad Zero \quad\ \to f \ m' \ n \\
&\qquad\qquad\qquad\qquad | \ Succ \ n' \to \textbf{case } (gt \ m \ n) \textbf{ of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad True \ \to f \ m' \ n \\
&\qquad\qquad\qquad\qquad\qquad\qquad | \ False \to f \ (Succ \ m) \ n' \\
&gt \ x \ y = \textbf{case } x \textbf{ of} \\
&\qquad\qquad Zero \quad\ \to False \\
&\qquad\quad\ | \ Succ \ x \to \textbf{case } y \textbf{ of} \\
&\qquad\qquad\qquad\qquad\qquad Zero \quad\ \to True \\
&\qquad\qquad\qquad\quad | \ Succ \ y \to gt \ x \ y
\end{aligned}
$$

In the function f, the first parameter both increases and decreases, so this causes problems for other termination checkers. The result of transforming this program using distillation is as follows:

$$
\begin{array}{l}
f\ m\ n \\
\textbf{where} \\
f\ m\ n = \textbf{case } m \textbf{ of} \\
\qquad\qquad Zero \quad\ \to Zero \\
\qquad\qquad |\ Succ\ m' \to \textbf{case } n \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad Zero \quad\ \to g\ m' \\
\qquad\qquad\qquad\qquad\qquad |\ Succ\ n' \to f\ m'\ n' \\
g\ m \quad = \textbf{case } m \textbf{ of} \\
\qquad\qquad Zero \quad\ \to Zero \\
\qquad\qquad |\ Succ\ m' \to g\ m'
\end{array}
$$

The LTS generated for this transformed program is shown in Figure 9. We can see that there is a case expression between every renamed state and its renaming, so this program is indeed terminating.

Figure 9: LTS Representation of Program in Example 5

Example 8 The final example shown in Figure 10 is McCarthy's 91 function, which is nested recursive and has often been used as a test case for proving termination. Although the result of transforming this program using distillation (and the corresponding LTS) are too large to show here, we are also able to prove the termination of this program.

6 Conclusion and Related Work

In this paper, we have described a new approach to the termination analysis of functional programs that is applied to the output of the *distillation* program transformation [15, 16]. Distillation converts programs into a simplified form called *distilled form*, and to prove that programs in this form terminate, we convert them into a corresponding *labelled transition system* and then show that all possible infinite *traces* through the labelled transition system would result in an infinite descent of well-founded data values.

```
f n
where
f n      = case (gt n (100)) of
                True  → sub n 10
                | False → f (f (plus n (11)))
gt x y   = case x of
                Zero   → False
                | Succ x → case y of
                                  Zero    → True
                                  | Succ y → gt x y
sub x y  = case y of
                Zero   → x
                | Succ y → case x of
                                  Zero    → Zero
                                  | Succ x → sub x y
plus x y = case x of
                Zero   → y
                | Succ x → Succ (plus x y)
```

Figure 10: McCarthy's 91 Function

We argue that our termination analysis is simple and straightforward. We do not need to treat nested function calls, mutual recursion or permuted arguments as special cases, we do not need to search for appropriate ranking functions and we do not need to define a size ordering on values. Most recent approaches to proving termination have involved searching for a set of possible ranking functions and using a disjunctive termination argument [3, 8, 19, 9]. We avoid the need for such involved analysis here.

The most closely related work to that described here is that described in [6] which makes use of *cyclic proof* techniques. In [6], a *cyclic pre-proof* form is defined that is a finite derivation tree in which every leaf that is not the conclusion of an axiom is closed by a backlink to a syntactically identical interior node. A global soundness condition is defined on pre-proofs so they can be verified as genuine *cyclic proofs*. This involves proving that every trace through the pre-proof is infinitely progressing and that there must therefore be infinite descent of the data values, in much the same way as is done in the work described here. The structure of a pre-proof is similar to the form of the labelled transition systems generated from programs that are in distilled form. However, a pre-proof does not contain any instances of the cut rule (which correspond to **let**s in distilled form) and therefore cannot have any accumulating parameters, so this technique is not applicable to as wide a range of programs as the technique described here. Also, because our programs have first been transformed into the form required to facilitate our proof, we are able to prove termination for an even wider class of programs.

Another closely related work to that described in this paper is the work on the *size-change principle* for termination [17]. In [17], *size-change graphs* are created that indicate definite information about the change of size of parameters in function calls. These graphs indicate whether a parameter is either decreasing or non-increasing. To prove termination of a program, it is then necessary to show that every possible thread within a program is *infinitely descending*, meaning that it contains infinitely many occurrences of a decreasing parameter. This is similar to the approach taken in this paper, where we also

try to show that there are infinitely many occurrences of a decreasing parameter. Both techniques can handle nested function calls. In [17], these are handled directly and in this work they are transformed to remove this nesting prior to analysis. However, there are also a number of differences between these two techniques. Firstly, in [17], if a parameter can possibly increase at any point in a thread, then it is not possible to determine whether it is infinitely descending. In this work, we can ignore this possibility as any parameter that decreases in size cannot previously have increased. However, in [17], a more precise measure of parameter size is employed based on their semantic value with a well-founded partial ordering. In this work, a more conservative measure of the syntactic size of parameters is used, where a parameter that is assigned to any expression other than a variable is considered to increase in size. In our running example, in the call *gcd (sub x y) y*, it is difficult to determine that the first parameter is semantically decreasing as we need to show that the number being subtracted is non-zero, so we cannot prove that it terminates using the size-change principle. Using our technique, even though we initially assume that this parameter is increasing as the syntactic size is increasing, the program is transformed by distillation to reveal that it is in fact decreasing, so we are able to prove termination. Also, our technique is directly applicable to higher-order languages, while the size-change principle originally described in [17] is not. An extension of the size-change principle to higher-order languages is described in [21], but it is far from straightforward. We have not been able to find any examples of programs that can be found to terminate using this size-change principle and cannot be found to terminate using the technique described here, but we have found many examples of the opposite being true. For example, none of the example programs in this paper can be shown to terminate using the size-change principle.

References

[1] Albert, E. and Arenas, P. and Genaim, S. and Puebla, G. and Zanardini, D. (2008): *COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode*. In: *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, Springer-Verlag, pp. 113—132 doi:10.1007/978-3-540-92188-2_5.

[2] T. Arts (1997): *Automatically Proving Termination and Innermost Normalisation of Term Rewriting Systems*. Ph.D. thesis, Universiteit Utrecht.

[3] J. Berdine, B. Cook, D. Distefano & P. W. O'Hearn (2006): *Automatic Termination Proofs for Programs with Shape-Shifting Heaps*. In: *International Conference on Computer Aided Verification*, pp. 386—400 doi:10.1007/11817963_35.

[4] A. Bradley, Z. Manna & H.B. Sipma (2005): *Linear Ranking with Reachability*. In: *17th International Conference on Computer Aided Verification,*, pp. 491–504 doi:10.1007/11513988_48.

[5] A. R. Bradley, Z. Manna & H. B. Sipma (2005): *Termination of Polynomial Programs*. In: *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 113–129 doi:10.1007/978-3-540-30579-8_8.

[6] J. Brotherston, R. Bornat & C. Calcagno (2008): *Cyclic Proofs of Program Termination in Separation Logic*. In: *ACM Symposium on Principles of Programming Languages*, pp. 101–112 doi:10.1145/1328897.1328453.

[7] M. Codish & C. Taboch (1997): *A Semantic Basis for Termination Analysis of Logic Programs and its Realization Using Symbolic Norm Constraints*. In: *International Joint Conference on Algebraic and Logic Programming, Lecture Notes in Computer Science* 1298, pp. 31–45 doi:10.1007/BFb0027001.

[8] B. Cook, A. Podelski & A. Rybalchenko (2006): *Termination Proofs for Systems Code*. *SIGPLAN Notices* 41(6), pp. 415—426 doi:10.1145/1133255.1134029.

[9] B. Cook, A. Podelski & A. Rybalchenko (2011): *Proving Program Termination*. Communications of the ACM 54(5), pp. 88—98 doi:10.1145/1941487.1941509.

[10] N. Dershowitz (1987): *Termination of Rewriting*. Journal of Symbolic Computation 3, pp. 69–116 doi:10.1016/S0747-7171(87)80022-6.

[11] R.W. Floyd (1967): *Assigning Meanings to Programs*. In: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19, pp. 19–32 doi:10.1007/978-94-011-1793-7_4.

[12] J. Geisl (1995): *Termination Analysis for Functional Programs Using Term Orderings*. In: *Second International Static Analysis Symposium, Lecture Notes in Computer Science* 983, pp. 154–171 doi:10.1007/3-540-60360-3_38.

[13] J. Giesl, M. Brockschmidt, F. Emmes, C. Frohn, F. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski & R Thiemann. (2014): *Proving Termination of Programs Automatically with AProVE*. In: *International Joint Conference on Automated Reasoning*, pp. 184–191 doi:10.1007/978-3-319-08587-6_13.

[14] J. Giesl, S. Swiderski, P. Schneider-Kamp & R. Thiemann (2006): *Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages*. In: *International Conference on Rewriting Techniques and Applications*, pp. 297–312 doi:10.1007/11805618_23.

[15] G.W. Hamilton (2007): *Distillation: Extracting the Essence of Programs*. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 61–70 doi:10.1145/1244381.1244391.

[16] G.W. Hamilton & N.D. Jones (2012): *Distillation With Labelled Transition Systems*. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM, pp. 15–24 doi:10.1145/2103746.2103753.

[17] C.S. Lee, N.D. Jones & A.M. Ben-Amram (2001): *The Size-Change Principle for Program Termination*. In: *The 28th ACM Symposium on Principles of Programming Languages*, pp. 81–92 doi:10.1145/360204.360210.

[18] N. Lindenstrauss & Y. Sagiv (1997): *Automatic Termination Analysis of Prolog Programs*. In: *International Conference on Logic Programming*, pp. 64–77.

[19] P. Manolios & D. Vroon (2006): *Termination Analysis with Calling Context Graphs*. In: *International Conference on Computer Aided Verification*, pp. 401–414 doi:10.1007/11817963_36.

[20] Y. Sagiv (1991): *A Termination Test for Logic Programs*. In: *International Symposium on Logic Programming*, pp. 518–532.

[21] D. Sereni D. & N.D. Jones (2005): *Termination Analysis of Higher-Order Functional Programs*. In: *Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science* 3780, pp. 281–297 doi:10.1007/11575467_19.

[22] M.H. Sørensen, R. Glück & N.D. Jones (1996): *A Positive Supercompiler*. Journal of Functional Programming 6(6), pp. 811–838 doi:10.1017/S0956796800002008.

[23] Spoto, F. and Mesnard, F. and Payet, E. (2010): *A Termination Analyzer for Java Bytecode Based on Path-Length*. ACM Transaction on Programming Languages and Systems 32(3), pp. 8:1–8:70 doi:10.1145/1709093.1709095.

[24] J. Steinbach (1995): *Automatic Termination Proofs With Transformation Orderings*. In: *International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science* 914, pp. 11–25 doi:10.1007/3-540-59200-8_44.

[25] V.F. Turchin (1986): *The Concept of a Supercompiler*. ACM Transactions on Programming Languages and Systems 8(3), pp. 90–121 doi:10.1145/5956.5957.

[26] A.M. Turing (1936): *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 2 42(1), pp. 230–265 doi:10.1112/plms/s2-42.1.230.

[27] A.M. Turing (1948): *Checking a Large Routine*. In: *The Early British Computer Conferences*, pp. 70–72 doi:10.5555/94938.94952.