Distilling Programs to Prove Termination

G.W. Hamilton

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

VPT 2020

Outline

1. Introduction
2. Language
3. Distillation
4. Labelled Transition Systems
5. Proving Termination
6. Examples
7. Conclusions

Introduction

- We consider the problem of determining whether a given program will always finish execution.
    - The halting problem famously shown to be undecidable by Turing (1936).
    - However, there are many programs for which termination can be proved.

Introduction

- We consider the problem of determining whether a given program will always finish execution.
  - The halting problem famously shown to be undecidable by Turing (1936).
  - However, there are many programs for which termination can be proved.
- The classic method for doing this dates back to Turing (1948):
  - Find a ranking function that maps a program state to a well-order
  - Prove that the result of this function decreases for every possible program transition.

Introduction

- We consider the problem of determining whether a given program will always finish execution.
  - The halting problem famously shown to be undecidable by Turing (1936).
  - However, there are many programs for which termination can be proved.
- The classic method for doing this dates back to Turing (1948):
  - Find a ranking function that maps a program state to a well-order
  - Prove that the result of this function decreases for every possible program transition.
- More recent approaches search for a set of ranking functions:
  - This set corresponds to a choice of ranking functions.
  - A disjunctive termination argument is therefore used.

Background

- Program termination techniques have been developed for functional programs:
  - e.g. Geisl, 1995; Lee, Jones and Ben-Amram, 2001; Manolios and Vroon. 2006

Background

- Program termination techniques have been developed for functional programs:
  - e.g. Geisl, 1995; Lee, Jones and Ben-Amram, 2001; Manolios and Vroon. 2006
- For logic programs:
  - e.g. Sagiv, 1991; Codish and Taboch, 1997; Lindenstrauss and Sagiv, 1997

Background

- Program termination techniques have been developed for functional programs:
  - e.g. Geisl, 1995; Lee, Jones and Ben-Amram, 2001; Manolios and Vroon. 2006
- For logic programs:
  - e.g. Sagiv, 1991; Codish and Taboch, 1997; Lindenstrauss and Sagiv, 1997
- And for imperative programs:
  - e.g. Floyd, 1967; Bradley, Manna and Sipma, 2005; Cook, Podelski and Rybalchenko, 2006

Approach

1. Apply distillation to the program being analysed.
    - produces a simplified form of program (distilled form) which is easier to analyse.
    - no special consideration needs to be given to nested function calls.

Approach

1. Apply distillation to the program being analysed.
   - produces a simplified form of program (distilled form) which is easier to analyse.
   - no special consideration needs to be given to nested function calls.
2. Convert transformed program into a corresponding labelled transition system.
   - abstracts away from function names and ordering of function arguments.
   - focuses on recursive structure of program.

Approach

1. Apply distillation to the program being analysed.
   - produces a simplified form of program (distilled form) which is easier to analyse.
   - no special consideration needs to be given to nested function calls.
2. Convert transformed program into a corresponding labelled transition system.
   - abstracts away from function names and ordering of function arguments.
   - focuses on recursive structure of program.
3. Analyse traces through labelled transition system.
   - show that within every cycle, at least one parameter is decreasing.
   - thus all possible infinite traces would result in an infinite descent of well-founded data values.

Language

Syntax

$prog ::= e_0 \text{ where } h_1 = e_1, \ldots, h_k = e_k$		Program
e	$::= x$	Variable
	$\mid c\ e_1 \ldots e_n$	Constructor Application
	$\mid \lambda x.e$	λ-Abstraction
	$\mid f$	Function Call
	$\mid e_0\ e_1$	Application
	$\mid \text{case } e_0 \text{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$	Case Expression
	$\mid \text{let } x = e_0 \text{ in } e_1$	Let Expression
h	$::= f\ x_1 \ldots x_n$	Function Header
p	$::= c\ x_1 \ldots x_n$	Pattern

Language

Semantics

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

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

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

$$(\text{let}\ x = e_0\ \text{in}\ e_1) \overset{\beta}{\rightsquigarrow} (e_1\{x \mapsto e_0\})$$

Example Program

$$
\begin{array}{l}
gcd\ x\ y \\
\text{where} \\
\begin{array}{lcl}
gcd\ x\ y & = & \text{case } x \text{ of} \\
& & \quad\ \ Zero \rightarrow y \\
& & \quad \mid Succ\ x' \rightarrow \text{case } y \text{ of} \\
& & \qquad\qquad\qquad\ \ Zero \rightarrow x \\
& & \qquad\qquad\qquad \mid Succ\ y' \rightarrow \text{case } (gt\ x\ y) \text{ of} \\
& & \qquad\qquad\qquad\qquad\qquad\quad\ \ True \rightarrow gcd\ (sub\ x\ y)\ y \\
& & \qquad\qquad\qquad\qquad\qquad\quad \mid False \rightarrow \text{case } (gt\ y\ x) \text{ of} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ True \rightarrow gcd\ x\ (sub\ y\ x) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid False \rightarrow x \\
gt\ x\ y & = & \text{case } x \text{ of} \\
& & \quad\ \ Zero \rightarrow False \\
& & \quad \mid Succ\ x' \rightarrow \text{case } y \text{ of} \\
& & \qquad\qquad\qquad\ \ Zero \rightarrow True \\
& & \qquad\qquad\qquad \mid Succ\ y' \rightarrow gt\ x'\ y' \\
sub\ x\ y & = & \text{case } y \text{ of} \\
& & \quad\ \ Zero \rightarrow x \\
& & \quad \mid Succ\ y' \rightarrow \text{case } x \text{ of} \\
& & \qquad\qquad\qquad\ \ Zero \rightarrow Zero \\
& & \qquad\qquad\qquad \mid Succ\ x' \rightarrow sub\ x'\ y'
\end{array}
\end{array}
$$

Distillation

Distillation converts programs into the distilled form form $e^{\emptyset}$ using distillation where e^{ρ} is defined as follows:

Distilled Form

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

let variables are added to the set ρ, and will not be used as case selectors, so the extracted expressions cannot be intermediate.

Example Program Distilled

$$
\begin{array}{l}
f0\ x\ y \\
\text{where} \\
f0\ a\ b \quad = \text{case}\ a\ \text{of} \\
\qquad\qquad Zero \quad \rightarrow b \\
\qquad\qquad \mid Succ\ a \rightarrow \text{case}\ b\ \text{of} \\
\qquad\qquad\qquad\qquad Zero \quad \rightarrow Succ\ a \\
\qquad\qquad\qquad\qquad \mid Succ\ b \rightarrow f1\ a\ b\ a\ b \\
f1\ a\ b\ c\ d = \text{case}\ a\ \text{of} \\
\qquad\qquad Zero \quad \rightarrow \text{case}\ b\ \text{of} \\
\qquad\qquad\qquad\qquad Zero \quad \rightarrow Succ\ c \\
\qquad\qquad\qquad\qquad \mid Succ\ b \rightarrow f1\ c\ b\ c\ b \\
\qquad\qquad \mid Succ\ a \rightarrow \text{case}\ b\ \text{of} \\
\qquad\qquad\qquad\qquad Zero \quad \rightarrow f1\ d\ a\ d\ a \\
\qquad\qquad\qquad\qquad \mid Succ\ b \rightarrow f1\ a\ b\ c\ d
\end{array}
$$

Labelled Transition Systems

A labelled transition system (LTS) is a 4-tuple $(\mathcal{E}, e_0, Act, \rightarrow)$ where:

Labelled Transition Systems

A labelled transition system (LTS) is a 4-tuple $(\mathcal{E}, e_0, Act, \rightarrow)$ where:

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

Labelled Transition Systems

A labelled transition system (LTS) is a 4-tuple $(\mathcal{E}, e_0, Act, \rightarrow)$ where:

- $\mathcal{E}$ is a set of states of the LTS. Each is an expression or the end-of-action state $\mathbf{0}$.
- $e_0 \in \mathcal{E}$ is the start state.

Labelled Transition Systems

A labelled transition system (LTS) is a 4-tuple $(\mathcal{E}, e_0, Act, \rightarrow)$ where:

- $\mathcal{E}$ is a set of states of the LTS. Each is an expression or the end-of-action state $\mathbf{0}$.
- $e_0 \in \mathcal{E}$ 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.

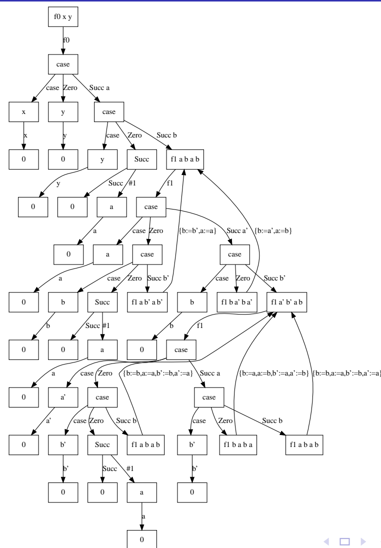
Labelled Transition Systems

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

Labelled Transition Systems

- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a transition relation. We write $e \stackrel{\alpha}{\rightarrow} e'$ for a transition from state e to state e' via action α.
- A folded LTS is a LTS which also contains renamings of the form $e \stackrel{\sigma}{\longrightarrow} e'$, where σ is a renaming s.t. $e \equiv e'\sigma$

Labelled Transition System For Distilled Example Program



Increasing and Decreasing Parameters

Definition (Decreasing Parameter)

A parameter is considered to decrease in size if it is the subject of a **case** selector.

Increasing and Decreasing Parameters

Definition (Decreasing Parameter)

A parameter is considered to decrease in size if it is the subject of a **case** selector.

Definition (Increasing Parameter)

A parameter is considered to increase in size if any expression other than a variable is assigned to it.

Increasing and Decreasing Parameters

Definition (Decreasing Parameter)

A parameter is considered to decrease in size if it is the subject of a **case** selector.

Definition (Increasing Parameter)

A parameter is considered to increase in size if any expression other than a variable is assigned to it.

Lemma (On Decreasing Parameters)

Every parameter that has decreased in size cannot previously have increased in size.

Definition (Trace)

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

Definition (Trace)

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

Definition (Infinitely Progressing Trace)

An infinitely progressing trace is a trace that contains an infinite number of decreases in parameter size.

Definition (Trace)

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

Definition (Infinitely Progressing Trace)

An infinitely progressing trace is a trace that contains an infinite number of decreases in parameter size.

Theorem (Termination)

If all traces through the labelled transition system generated from the result of distilling a program are infinitely progressing, then the program terminates.

Example

Consider the following program:

$$
\begin{array}{l}
f\ n \\
\text{where} \\
f\ n = \text{case}\ n\ \text{of} \\
\qquad\qquad \begin{array}{ll} Zero & \rightarrow Zero \\ \mid Succ\ n' & \rightarrow g\ (Succ\ n) \end{array} \\
g\ n = \text{case}\ n\ \text{of} \\
\qquad\qquad \begin{array}{ll} Zero & \rightarrow Zero \\ \mid Succ\ n' & \rightarrow \text{case}\ n'\ \text{of} \\ & \qquad \begin{array}{ll} Zero & \rightarrow Zero \\ \mid Succ\ n'' & \rightarrow f\ n'' \end{array} \end{array}
\end{array}
$$

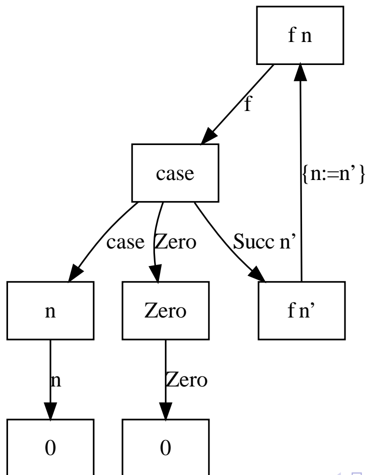
Example

The result of transforming this program using distillation is as follows:

$$
\begin{array}{l}
f\ \ n \\
\text{where} \\
f\ \ n = \text{case}\ n\ \text{of} \\
\qquad\qquad\quad \begin{array}{ll} \ \ Zero & \rightarrow Zero \\ \mid Succ\ \ n' & \rightarrow f\ \ n' \end{array}
\end{array}
$$

Example

The LTS generated for this transformed program is as follows:



Example

Consider the following program:

$$
\begin{array}{l}
f\ m\ n \\
\text{where} \\
\begin{array}{lcl}
f\ m\ n & = & \text{case } m \text{ of} \\
 & & \quad \begin{array}{ll} \ \ Zero & \rightarrow Zero \\ \mid Succ\ m' & \rightarrow f\ (sub\ m\ n)\ (Succ\ n) \end{array} \\
sub\ x\ y & = & \text{case } y \text{ of} \\
 & & \quad \begin{array}{ll} \ \ Zero & \rightarrow x \\ \mid Succ\ y & \rightarrow \text{case } x \text{ of} \\ & \qquad \begin{array}{ll} \ \ Zero & \rightarrow Zero \\ \mid Succ\ x & \rightarrow sub\ x\ y \end{array} \end{array}
\end{array}
\end{array}
$$

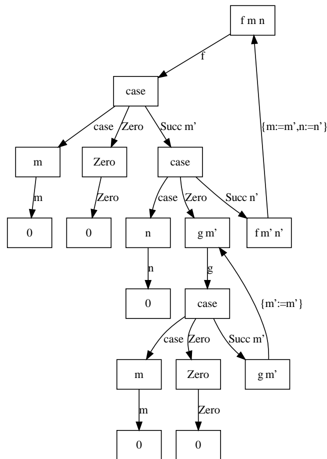
Example

The result of transforming this program using distillation is as follows:

$$
\begin{array}{l}
f\ \ m\ \ n \\
\text{where} \\
\begin{array}{lcl}
f\ \ m\ \ n & = & \text{case}\ m\ \text{of} \\
 & & \quad \begin{array}{ll} \ \ Zero & \rightarrow Zero \\ \mid Succ\ \ m' & \rightarrow \text{case}\ n\ \text{of} \\ & \quad \begin{array}{ll} \ \ Zero & \rightarrow g\ \ m' \\ \mid Succ\ \ n' & \rightarrow f\ \ m'\ \ n' \end{array} \end{array} \\
g\ \ m & = & \text{case}\ m\ \text{of} \\
 & & \quad \begin{array}{ll} \ \ Zero & \rightarrow Zero \\ \mid Succ\ \ m' & \rightarrow g\ \ m' \end{array}
\end{array}
\end{array}
$$

Example

The LTS generated for this transformed program is as follows:



Example

Consider the following program:

$$
\begin{array}{l}
f\ m\ n \\
\text{where} \\
f\ m\ n = \text{case } m \text{ of} \\
\qquad\qquad Zero \quad \rightarrow Zero \\
\qquad\quad\ \mid Succ\ m' \rightarrow \text{case } n \text{ of} \\
\qquad\qquad\qquad\qquad\qquad Zero \quad \rightarrow f\ m'\ n \\
\qquad\qquad\qquad\qquad\quad\ \mid Succ\ n' \rightarrow \text{case } (gt\ m\ n) \text{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad True \ \rightarrow f\ m'\ n \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid False \rightarrow f\ (Succ\ m)\ n' \\
gt\ x\ y = \text{case } x \text{ of} \\
\qquad\qquad Zero \quad \rightarrow False \\
\qquad\quad\ \mid Succ\ x \rightarrow \text{case } y \text{ of} \\
\qquad\qquad\qquad\qquad Zero \quad \rightarrow True \\
\qquad\qquad\qquad\quad\ \mid Succ\ y \rightarrow gt\ x\ y
\end{array}
$$

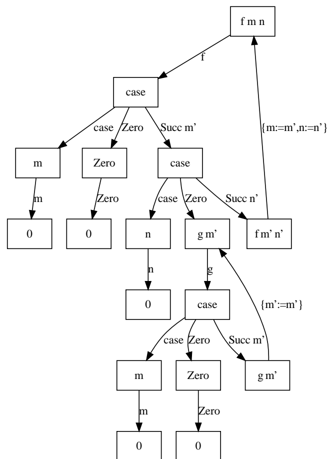
Example

The result of transforming this program using distillation is as follows:

$$
\begin{array}{l}
f\ \ m\ \ n \\
\text{where} \\
f\ \ m\ \ n = \text{case}\ \ m\ \ \text{of} \\
\qquad\qquad\quad Zero \quad\ \ \rightarrow Zero \\
\qquad\qquad\ \ \mid Succ\ \ m' \rightarrow \text{case}\ \ n\ \ \text{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad\ \ \rightarrow g\ \ m' \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \mid Succ\ \ n' \rightarrow f\ \ m'\ \ n' \\
g\ \ m \quad = \text{case}\ \ m\ \ \text{of} \\
\qquad\qquad\quad Zero \quad\ \ \rightarrow Zero \\
\qquad\qquad\ \ \mid Succ\ \ m' \rightarrow g\ \ m'
\end{array}
$$

Example

The LTS generated for this transformed program is as follows:



Conclusions

- We have described a new approach to the termination analysis of functional programs.
  - Distillation converts programs into a simplified form called distilled form.
  - The resulting programs are converted into corresponding labelled transition systems.
  - It is shown that all possible infinite traces through the labelled transition system would result in an infinite descent of well-founded data values.

Conclusions

- We have described a new approach to the termination analysis of functional programs.
  - Distillation converts programs into a simplified form called distilled form.
  - The resulting programs are converted into corresponding labelled transition systems.
  - It is shown that all possible infinite traces through the labelled transition system would result in an infinite descent of well-founded data values.
- 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.
  - We do not need to define a size ordering on values.

Related Work

- Cyclic proof techniques (Brotherston, Bornat and Calcagno, 2008):
  - Cyclic pre-proof corresponds roughly to our labelled transition systems.
  - Does not allow for accumulating parmeters.
  - We transform programs into the required form, so are able to prove termination for an even wider class of programs.

Related Work

- Cyclic proof techniques (Brotherston, Bornat and Calcagno, 2008):
  - Cyclic pre-proof corresponds roughly to our labelled transition systems.
  - Does not allow for accumulating parmeters.
  - We transform programs into the required form, so are able to prove termination for an even wider class of programs.
- Size-change principle (Lee, Jones and Ben-Amram, 2001):
  - It needs to be shown that every possible thread within a program is infinitely descending.
  - If a parameter can possibly increase at any point in a thread it is not possible to determine whether it is infinitely descending.
  - A more precise measure of parameter size is used based on their semantic value with a well-founded partial ordering.