Bachelor thesis
Computer Science

Radboud University

Characteristic Formulas and CakeML

Author:
Joris den Elzen
s4481038

First supervisor/assessor:
dr. Freek Wiedijk
`freek@cs.ru.nl`

Second assessor:
prof. dr. Herman Geuvers
`H.Geuvers@cs.ru.nl`

December 12, 2018

Abstract

In this thesis, we examine and demonstrate the use of characteristic formulae for program verification using automated theorem provers/proof assistants. In particular, we examine the implementation of characteristic formulae for the CakeML language. CakeML is a subset of the functional programming language Standard ML, with a verified compiler backend and a range of tools for verification proofs, implemented in the HOL4 theorem prover. The formal definitions presented in this thesis are based on those developed by Arthur Chargueraud in his thesis [8]. The thesis aims to provide a clear overview of the theory and practise of using characteristic formulae for program verification proofs.

Contents

Acknowledgements

I would like to thank my supervisor dr. Freek Wiedijk for his patience and involved assistance in writing (and in particular finishing) this thesis. I would also like to acknowledge the help provided by prof. dr. Herman Geuvers for coming up with the substitution in the characteristic formula. Assistance provided by Johannes Åman Pohjola and Magnus Myreen in the CakeML Slack channel was greatly appreciated when HOL4 was giving me headaches.

Introduction

The traditional way of verifying software using a mechanized proof is to manually write a logical representation of the behaviour of a program, and to prove that this representation satisfies a certain specification in the same logic. Using characteristic formulas, it is possible to compute a representation of a function that describes the behaviour of a piece of code as a formula in higher-order logic. This eliminates the need to manually translate code to logic, thus reducing workload of the verification process and human error. The CakeML [12] programming and verification toolset contains an implementation of this theory for a subset of ML (metalanguage).

In this thesis we research the viability of CakeML's program verification method. In order to do this we aim to produce a fully verified implementation of a simple function `double`. The function simply takes an integer as input and doubles it. We will first go through the entire process on paper. Afterwards we will go through the same process in HOL4, to demonstrate how the theory is to be used in practice. The process of implementing and verifying a program using CakeML is documented from start to finish, including installation instructions to replicate the development setup used for the project. We aim to elaborate on the link between the formal definitions of characteristic formulas as defined by Charguéraud [8] and the CF implementation of CakeML [10] in the HOL4 theorem prover [1].

I would like to than

Chapter 1

Background

1.1 Mechanized software verification proofs

The traditional way of verifying software using a mechanized proof is to manually write a logical embedding of a program, and to prove that this representation satisfies a certain specification in the same logic. Using characteristic formulas, it is possible to compute a representation of a function that describes the behaviour of a piece of code as a formula in higher-order separation logic. This eliminates the need to manually translate code to logic, thus reducing workload of the verification process and human error.

1.2 HOL

1.2.1 Overview

HOL4 is an interactive theorem prover for higher order logic. It is a programmable environment for specifying and proving theorems. HOL is programmed in SML (Standard ML). HOL4 is only one of four major HOL-type theorem provers, all with essentially the same basic logic.

Other interactive theorem provers in the HOL-family are HOL Light [5], ProofPower [4], HOL Zero [2] and Isabelle/HOL [3]. HOL Light started out as a minimalistic implementation effort of HOL, but has since grown to a major theorem prover with many supporting libraries, even though its core code is still modestly simple.

The precursor to all HOL-systems is LCF [9]. LCF is a proof-checking program developed at Stanford University in 1972 by Robert Milner. Milner also designed the programming language ML underlying both LCF and modern day HOL. Notable members of the original LCF team include Whitfield Diffie, now well-known in cryptography. An early problem of LCF was that proofs were stored as data structures representing the proof, which consumed a significant amount of memory. To repair this, Milner devised a representation of theorems as an abstract data type, which had axioms as predefined instances and inference rules as operations over the datatype. Strict type checking ensured that the values of this type could only be created when they could be obtained from the axioms by applying the inference rules. This idea of theorems as an abstract datatype is fundamental to modern day HOL.

Milner designed ML in order to make the set of commands used in LCF easily extendable, and ML was strictly typed in order to support the theorem datatype. ML was made as a functional programming language so that subgoaling strategies (named “tactics” by Milner, they are still called tactics in HOL) could be implemented as functions and combining tactics could be implemented as higher-order functions taking tactics as argument, and returning them as a result. Standard ML design was strongly influenced by the needs of theorem proving, which is why it, and its derivative languages (such as OCaml), remains a standard for theorem prover implementation today.

Development of HOL started at Cambridge by Mike Gordon, when he was using LCF for hardware and started making his own alterations and additions to the program. HOL88 was the eventual result of the development process started by Mike Gordon. HOL88 was developed alongside its own version of ML, which was implemented in Common Lisp. Further iterations of HOL88 (HOL90, HOL98 and HOL4) were all implemented in Standard ML. HOL4 is the latest iteration and is the only version that is still being maintained.

1.2.2 Installation

For the case study HOL4 was used alongside the text-editor emacs, which made the specification of theorems and the construction of proofs very convenient. This section will explain how to replicate the setup used in the case study. It is assumed you are working on a Unix-like operating system such as Linux or MacOS.

First HOL4 needs to be installed. The program will have to be compiled on your machine. You can download the code from github:

```
git clone git://github.com/HOL-Theorem-Prover/HOL.git
```

After cloning the repository you will have a directory HOL containing all the current source files.

HOL4 is built in Standard ML. Standard ML currently has two major implementations: Moscow ML and Poly/ML. For the case study Poly/ML was used, as it is the preferred implementation of ML for building large projects such as HOL4. The source tarball can be downloaded here:

```
https://github.com/polyml/polyml/releases
```

Extract the tarball and build Poly/ML:

```
./configure
make
make install
```

Now that you have an SML compiler, HOL can be built:

```
polyml < tools/smart-configure.sml
bin/build
```

For making interaction with HOL significantly easier it is recommended to install the `emacs` text editor for your distribution. This can probably be done through your distribution's package manager. Add this line to your `emacs` init file (`~/.emacs` or `~/.emacs.d/init.el`):

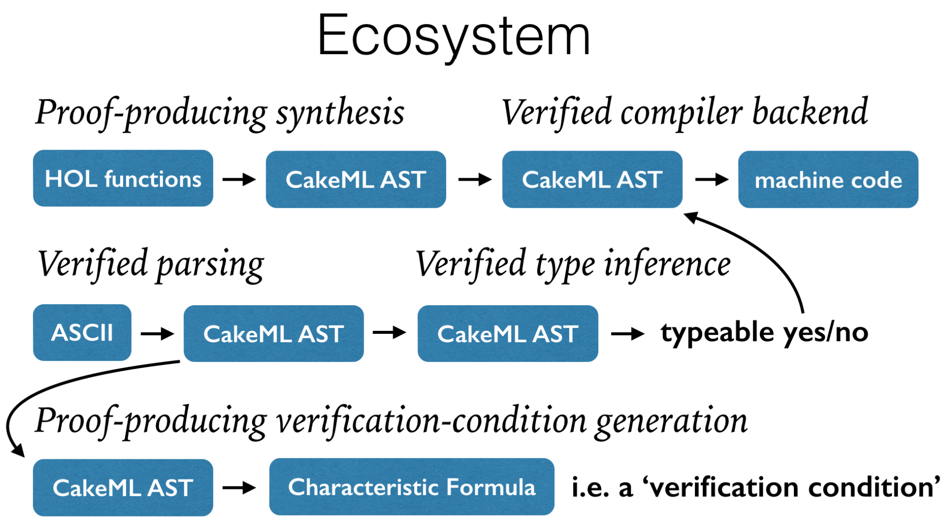
```
(load "<path>/HOL/tools/hol-mode")
```

A guide by Magnus O. Myreen[14] further elaborates on the use of `emacs` with HOL4.

1.3 CakeML

1.3.1 Overview

The implementation of CF studied in this thesis is the implementation of CakeML [18][12][17][10][15][16]. CakeML is a functional programming language that is a subset of Standard ML, with an environment of tools for program verification.



The most significant part of the CakeML ecosystem is the verified compiler backend[17] which takes a CakeML abstract syntax tree and produces machine code for 5 different architectures: x86-64, ARMv6, ARMv8, MIPS-64, and RISC-V. The compiler passes through a total of 12 intermediate language, abstracting away high level features, in order for verification to take place at the right levels of semantic detail. It has been developed entirely within the HOL4 theorem prover.

The CakeML compiler has two different frontends, one of which is the translator [15], which generates CakeML abstract syntax trees from HOL4 specifications in higher order logic. The translator tool also proves that the generated CakeML behaves the same as the HOL4 function. This theorem can then be used with proofs of claims in the logic, which transfers to the program through this generated theorem. The other frontend is a verified parser that generates ASTs from CakeML source code.

This thesis focuses on the CakeML implementation of Chargueraud's CFML verification framework. This is a verification framework using characteristic formulas for the Caml programming language.

1.3.2 Grammar

CakeML is syntactically a subset of Standard ML. Its grammar is described below, as taken from [12]:

$$
\begin{aligned}
\mathit{id} &:= x \mid \mathit{Mn}.\,x \\
\mathit{cid} &:= \mathit{Cn} \mid M.\,\mathit{Cn} \\
t &:= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \alpha \mid \mathit{id} \mid t\ \mathit{id} \mid \texttt{(}t(\texttt{,}t)^*\texttt{)}\,\mathit{id} \\
&\quad \mid t\texttt{*}t \mid t\ \texttt{->}\ t \mid t\ \texttt{ref} \mid \texttt{(}t\texttt{)} \\
l &:= i \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{[]} \\
p &:= x \mid l \mid \mathit{cid} \mid \mathit{cid}\ p \mid \texttt{ref}\ p \mid \texttt{\_} \mid \texttt{(}p(\texttt{,}p)^*\texttt{)} \mid \texttt{[}p(\texttt{,}p)^*\texttt{]} \\
&\quad \mid p\texttt{::}p \\
e &:= l \mid \mathit{id} \mid \mathit{cid} \mid \mathit{cid}\ e \mid \texttt{(}e\texttt{,}e(\texttt{,}e)^*\texttt{)} \mid \texttt{[}e(\texttt{,}e)^*\texttt{]} \\
&\quad \mid \texttt{raise}\ e \mid e\ \texttt{handle}\ p\ \texttt{=>}\ e(\texttt{|}p\texttt{=>}e)^* \\
&\quad \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e\ e \mid \texttt{((}e\texttt{;})^*e\texttt{)} \mid \mathit{uop}\ e \mid e\ \mathit{op}\ e \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{case}\ e\ \texttt{of}\ p\ \texttt{=>}\ e(\texttt{|}p\texttt{=>}e)^* \\
&\quad \mid \texttt{let}\ (\mathit{ld} \mid \texttt{;})^*\ \texttt{in}\ (e\texttt{;})^*\ e\ \texttt{end} \\
\mathit{ld} &:= \texttt{val}\ x\ \texttt{=}\ e \mid \texttt{fun}\ x\ y^+ \texttt{=} e\ (\texttt{and}\ x\ y^+\ \texttt{=}\ e)^* \\
\mathit{uop} &:= \texttt{ref} \mid \texttt{!} \mid \\
\mathit{op} &:= \texttt{=} \mid \texttt{:=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{div} \mid \texttt{mod} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<>} \mid \texttt{::} \\
&\quad \mid \texttt{before} \mid \texttt{andalso} \mid \texttt{orelse} \\
c &:= \mathit{Cn} \mid \mathit{Cn}\ \texttt{of}\ t \\
\mathit{tyd} &:= \mathit{tyn}\ \texttt{=}\ c(\texttt{|}c)^* \\
\mathit{tyn} &:= \texttt{(}\alpha(\texttt{,}\alpha)^*\texttt{)}\ x \mid \alpha\,x \mid x \\
d &:= \texttt{datatype}\ \mathit{tyd}\ (\texttt{and}\ \mathit{tyd})^* \mid \texttt{val}\ p\ \texttt{=}\ e \\
&\quad \mid \texttt{fun}\ x\,y^+\ \texttt{=}\ e\ (\texttt{and}\ x\,y^+\ \texttt{=}\ e)^* \\
&\quad \mid \texttt{exception}\ c \\
\mathit{sig} &:= \texttt{:>}\,\texttt{sig}\ (\mathit{sl} \mid \texttt{;})^*\ \texttt{end} \\
\mathit{sl} &:= \texttt{val}\ x\ \texttt{:}\ t \mid \texttt{type}\ \mathit{type} \mid \texttt{datatype}\ \mathit{tyd}\ (\texttt{and}\ \mathit{tyd})^* \\
\mathit{top} &:= \texttt{structure}\ \mathit{Mn}\ \mathit{sig}^?\ \texttt{=}\ \texttt{struct}\ (d \mid \texttt{;})^*\ \texttt{end;} \mid d\texttt{;} \mid e\texttt{;}
\end{aligned}
$$

where x and y range over identifiers (must not start with a capital letter), α over SML-style type variables (e.g. `'a`), Cn over constructor names (must start with a capital letter), Mn over module names, and i over integers.

1.3.3 Formalization

CakeML's semantics are defined using so-called functional big-step semantics [16]. In contrast to conventional big-step semantics, these are defined by recursive equations rather than inductively defined relations. Initially CakeML semantics were defined using a combination of big-step and small-step operational semantics. A functional big-step semantics is essentially an purely functional interpreter equipped with a clock, to ensure totality of the interpreter even if it is run on a diverging program. The definition of the semantics can be found in the CakeML repository [18].

1.3.4 Installation

You can get CakeML by cloning the repository:

```
git clone https://github.com/CakeML/cakeml
```

And you can build it using `Holmake`:

```
cd cakeml
~/HOL/bin/Holmake
```

This does require a lot of RAM ($\sim$16 GB) and takes a very long time ($\sim$20 hours) so I recommend only building the parts you need. Running `Holmake` in the directory `cakeml/characteristic/examples` is enough for following along with the examples in this thesis.

Chapter 2

Characteristic Formulas

2.1 Overview

The concept of characteristic formulas originates in concurrency theory, as the characterization theorem of bisimulation equivalence [7]. Developed by Mathew Hennessy and Robin Milner [11], this theorem states that two bisilimar processes are equivalent if and only if they satisfy the same formula in Hennessy-Milner logic. Bisimilarity is a notion to describe behavioural equivalence of processes, and Hennessy-Milner logic is a logic used to specify labelled transition systems. Other iterations include Graf and Sifakis' translation from a term in CCS [6], Milner's calculus of communicating systems [13], to a formula in a modal language that represents an equivalence class with respect to observational congruence (a weaker definition of equivalence than bisimilar equivalence) of terms in CCS.

A characteristic formula, in the context of this thesis, is a higher-order logical formula that describes the behaviour of the program it is generated from. These types of characteristic formulas have been developed by Arthur Charguéraud for the Caml language. Charguéraud's work includes software to generate characteristic formulas for Caml programs in Coq. These formulas have been proven sound and correct (on paper) with the respect to the semantics of the language. The software studied in this thesis is the CakeML implementation of Charguéraud's work, since in combination with the other CakeML tools, such as the verified compiler and the translation tool, this provides an elegant framework for verified programming. Characteristic formulas are generated by a recursive function that generates the formula top-down. In order to illustrate how this work, we will define a very simple functional programming language, along with a definition of the generation function over this language.

2.2 Example language

For illustration we will use a very simple language containing variables, integers, booleans and basic constructions such as let and if. For simplification purposes we do not define a type system for this language, but we assume that some type system exist. In our proofs we type all program values as 'MLVal' types. Although the language does not contain unbounded lambda expressions, every lambda expression can be constructed: $(\lambda x.t)\ v \equiv \text{let rec } f\ =\ \lambda x.t \text{ in } f\, v$.

$$
\begin{aligned}
x, f &\coloneqq \textit{variables} \\
n &\coloneqq \textit{integers} \\
b &\coloneqq \{\textit{true}, \textit{false}\} \\
v &\coloneqq x \mid n \mid b \mid \mu f.\lambda x.t \\
t &\coloneqq v \mid (f\ t) \mid \text{if } t \text{ then } t \text{ else } t \mid \text{let } x = t \text{ in } t \mid \text{let rec } f = \lambda x.t \text{ in } t
\end{aligned}
$$

Before a characteristic formula is generated, a program is reduced to a 'normal form', where function arguments and if-statement conditions are bound to let-expressions. The normal form ensures an evaluation order that makes it possible for characteristic formulas to abstract away from these terms, making it possible to prove something general about the term and using this in the rest of the proof, when this term is replaced by a quantification. Take the characteristic formula for the let-rule:

$$[\![\text{let } x = t_1 \text{ in } t_2]\!] \equiv \lambda P.\, \exists P'.\, [\![t_1]\!] P' \wedge (\forall x.\, P' x \Rightarrow [\![t_2]\!] P)$$

Here P and P' are postconditions, predicates of type MLVal $\rightarrow$ Prop, Prop being the type of logical propositions, and MLVal is the type of some value in the language. When proving this characteristic formula, you instantiate P' as some predicate, prove that this predicate is true for t_1, and then abstract away from this term, using $P'\,x$ as an abstract representation of what t_1 does. This makes proving the characteristic formulas of if-statements easier; we first prove whether t_1 is true or false, and abstract away from the term, using the fact that it evaluates to either true or false. The normal form is given by the function *nf*:

$$\begin{aligned}
\mathit{nf}(v) &\equiv v \\
\mathit{nf}(f\ t) &\equiv \text{let } x = \mathit{nf}(t) \text{ in } f\ x \\
&\ \text{with } x \text{ a fresh variable} \\
\mathit{nf}(\text{if } t_c \text{ then } t_t \text{ else } t_e) &\equiv \text{let } c = t_c \text{ in } (\text{if } c \text{ then } \mathit{nf}(t_t) \text{ else } \mathit{nf}(t_e)) \\
&\ \text{with } c \text{ a fresh variable} \\
\mathit{nf}(\text{let } x = t_1 \text{ in } t_2) &\equiv \text{let } x = \mathit{nf}(t_1) \text{ in } \mathit{nf}(t_2) \\
\mathit{nf}(\text{let rec } f = \lambda x.t_1 \text{ in } t_2) &\equiv \text{let rec } f = \lambda x.\mathit{nf}(t_1) \text{ in } \mathit{nf}(t_2)
\end{aligned}$$

The grammar of programs in normal form is then:

$$\begin{aligned}
x, f &\coloneqq \mathit{variables} \\
n &\coloneqq \mathit{integers} \\
b &\coloneqq \{\mathit{true}, \mathit{false}\} \\
v &\coloneqq x \mid n \mid b \mid \mu f.\lambda x.t \\
t &\coloneqq v \mid (f\ v) \mid \text{if } v \text{ then } t \text{ else } t \mid \text{let } x = t \text{ in } t \mid \text{let rec } f = \lambda x.t \text{ in } t
\end{aligned}$$

Function closures ($\mu f.\lambda x.t$) are used to represent functions as a value. These are not found in actual program code but are used in the natural semantics rules. The semantics (for closed programs):

$$\frac{}{v \Downarrow v}\ \textsc{Val} \qquad \frac{([f \to \mu f.\lambda x.t_1][x \to v_2]t_1) \Downarrow v}{((\mu f.\lambda x.t_1)v_2) \Downarrow v}\ \textsc{App}$$

$$\frac{t_1 \Downarrow v_1 \qquad [x \to v_1]t_2 \Downarrow v}{(\text{let } x = t_1 \text{ in } t_2) \Downarrow v}\ \textsc{Let} \qquad \frac{([f \to \mu f.\lambda x.t_1]t_2) \Downarrow v}{(\text{let rec } f = \lambda x.t_1 \text{ in } t_2) \Downarrow v}\ \textsc{Rec}$$

$$\frac{t_1 \Downarrow v}{\text{if } \mathit{true} \text{ then } t_1 \text{ else } t_2 \Downarrow v}\ \textsc{IfT} \qquad \frac{t_2 \Downarrow v}{\text{if } \mathit{false} \text{ then } t_1 \text{ else } t_2 \Downarrow v}\ \textsc{IfF}$$

The substitution operation ρt, where the domain of ρ only contains free variables in t, is defined as:

$$
\begin{aligned}
\rho x &= \begin{cases} v' & \text{if } (x \mapsto v') \in \rho \\ x & \text{otherwise} \end{cases} \\
\rho n &= n \\
\rho b &= b \\
\rho(\mu f.\lambda x.t) &= \mu f.\lambda x.\rho t \\
\rho(f\, v) &= (\rho f)\,(\rho v) \\
\rho(\text{let } x = t_1 \text{ in } t_2) &= \text{let } x = \rho t_1 \text{ in } \rho t_2 \\
\rho(\text{let rec } f = \lambda x.t_1 \text{ in } t_2) &= \text{let rec } f = \lambda x.\rho t_1 \text{ in } \rho t_2 \\
\rho(\text{if } v \text{ then } t_1 \text{ else } t_2) &= \text{if } v \text{ then } \rho t_1 \text{ else } \rho t_2
\end{aligned}
$$

where x is a variable, n is an integer and b is a boolean.

2.3 Definition of CF

Now that we have established a small example language, we can define the function that calculates characteristic formulas. The characteristic formula of a term t is notated as $[\![t]\!]$. P denotes the postcondition, a predicate of type MLVal $\rightarrow$ Prop, Prop being the type of logical propositions, and MLVal being the type of program values.

$$
\begin{aligned}
[\![v]\!]_\rho &\equiv \lambda P.\, P\, \langle v \rangle_\rho \\
[\![f\ v]\!]_\rho &\equiv \lambda P.\, \text{AppReturns}\, \langle f \rangle_\rho\, \langle v \rangle_\rho\, P \\
[\![\text{if } v \text{ then } t_1 \text{ else } t_2]\!]_\rho &\equiv \lambda P.\, (\langle v \rangle_\rho = \text{true} \Rightarrow [\![t_1]\!]_\rho P) \wedge (\langle v \rangle_\rho = \text{false} \Rightarrow [\![t_2]\!]_\rho P) \\
[\![\text{let } x = t_1 \ \text{ in } \ t_2]\!]_\rho &\equiv \lambda P.\, \exists P'.\, [\![t_1]\!]_\rho P' \wedge (\forall X.\, P' X \Rightarrow [\![t_2]\!]_{(\rho, x \mapsto X)} P) \\
[\![\text{let rec} f = \lambda x.\, t_1 \text{in} t_2]\!]_\rho &\equiv \lambda P.\, \forall F. \\
&\quad (\forall X.\, \forall P'.\, [\![t_1]\!]_{(\rho, f \mapsto F, x \mapsto X)} P' \Rightarrow \text{AppReturns}\, F\, X\, P') \\
&\quad \Rightarrow [\![t_2]\!]_{(\rho, f \mapsto F)} P
\end{aligned}
$$

The characteristic formula also takes a substitution ρ, to take care of the quantifications over variables in the code. For $[\![t]\!]_\rho$, the domain of ρ only contains free variables in t, it cannot contain any variables bound in t (this also follows from the definition of $[\![]\!]$). This substitution is applied as such, where x is a variable, n an integer and b a boolean:

$$\begin{aligned}\langle x\rangle_\rho &= \rho x\\ \langle n\rangle_\rho &= n\\ \langle b\rangle_\rho &= b\\ \langle \mu f.\lambda x.t\rangle_\rho &= \mu f = \lambda x.\rho t\end{aligned}$$

2.4 Definition of AppReturns

In Chargueraud's CFML the predicate AppReturns is used to specify functions. Because program functions can fail or diverge, and logical functions cannot, an abstract type Func is needed to describe program functions in the logic. This type can be viewed as the set of all function closures. Values of this type are specified using the predicate AppReturns. Informally, the proposition AppReturns $f\ x\ P$ states that a function f applied on argument x returns a value satisfying P. The type of AppReturns is then:

$$\text{AppReturns} : \forall AB.\,\text{Func} \to A \to (B \to \text{Prop}) \to \text{Prop}$$

Prop is the type of propositions, expressions that can be true or false. AppReturns is defined in terms of the predicate AppEval. AppEval is one of three axioms that the CFML library is built on. The proposition AppEval $f\ v\ v'$ states that the application of f on v produces the value v'. The type of AppEval is:

$$\text{AppEval} : \forall AB.\,\text{Func} \to A \to B \to \text{Prop}$$

Using AppEval we can intuitively define AppReturns:

$$\text{AppReturns}\ f\ v\ P\ \equiv \exists v'.\ \text{AppEval}\ f\ v\ v' \wedge P\ v'$$

which that the application of f over v produces a values v' satisfying P. Chargueraud gives an interpretation of the AppEval predicate in his soundness proof of CFML. In our example we work with a very small example language and abstract higher order logic and thus we are disregarding the translation that has to take place between Caml and Coq values. The formal interpretation of AppEval then comes down to the following proposition:

$$\text{AppEval}\ f\ v\ v'\ \equiv\ f\ v \Downarrow v'$$

2.5 Soundness and completeness

In this section the defined characteristic formulas will be proven sound and complete with respect to the defined big-step semantics of the example language. Before we start we need to prove some preliminary lemmas.

Lemma 3.1.1 (Substitution lemma for values) Let v' be a value in the normal form language, let v be some value and let ρ be a substitution not containing y, and only containing free variables in v. Then:

$$\langle [y \mapsto v]v' \rangle_\rho = \langle v' \rangle_{(\rho, y \mapsto v)}$$

Proof For values that are not the variable y or function closures the proposition trivially holds. If v' is the variable y, then:

$$\begin{aligned}
&\langle y \rangle_{(\rho, y \mapsto v)} \\
=&[\rho, y \mapsto v](y) \\
=&v \\
=&\langle v \rangle_\rho \\
=&\langle [y \mapsto v]y \rangle_\rho
\end{aligned}$$

If v' is a closure $\mu f = \lambda x.t$:

$$\begin{aligned}
&\langle \mu f = \lambda x.t \rangle_{(\rho, y \mapsto v)} \\
=&(\mu f = \lambda x.[\rho, y \mapsto v]t) \\
=&\mu f = \lambda x.\rho([y \mapsto v]t) \\
=&\langle \mu f = \lambda x.[y \mapsto v]t \rangle_\rho \\
=&\langle [y \mapsto v](\mu f = \lambda x.t) \rangle_\rho
\end{aligned}$$

Lemma 3.1.2 (Substitution lemma) Let t be a term in normal form with a free variable x, let v be some value and let ρ be a substitution not containing x. Then:

$$[\![[x \mapsto v]t]\!]_\rho = [\![t]\!]_{(\rho, x \mapsto v)}$$

Proof With induction over the structure of t. When we encounter a value we apply Lemma 3.1.1.

Soundness says that if the characteristic formula of a term t holds for a some postcondition P, then t will reduce to some value r for which P holds, according to the big-step semantics. It essentially means that the characteristic formula describes the corresponding program correctly. The soundness of our CF definition is described by the following theorem:

Theorem 3.2 For every term t, postcondition P and substitution ρ (where ρ contains the free variables in t) it holds that, if the characteristic formula of t satisfies P then t reduces to some value r for which P holds. Formally:

$$\forall t.\, \forall P.\, \forall \rho.\, [\![t]\!]_\rho\, P \implies \exists r.\, \rho t \Downarrow r \wedge P\, r$$

We prove this theorem by induction over the structure of t, proving that this theorem holds for every CF rule, starting with the non-recursive rules. Since the CF corresponds closely to the semantics, this works quite intuitively.

Proof: We prove theorem 2.1 by induction over the structure of t.

case $[\![v]\!]_\rho$: Assume $[\![v]\!]_\rho P$, then $P\, \langle v \rangle_\rho$ holds, and by the first substitution lemma $P\,(\rho v)$ holds. According to semantics rule VAL, $(\rho v) \Downarrow (\rho v)$. Then $\exists r.\, (\rho v) \Downarrow r \wedge Pr$.

case $[\![f v]\!]_\rho$: Assume $[\![f v]\!]_\rho P$. Then AppReturns $\langle\, f \rangle_\rho\, \langle v \rangle_\rho\, P$ gives us AppReturns $(\rho f)\, (\rho\, v)\, P$ by the first substitution lemma. From the definition of AppReturns we get that there exists a v' such that AppEval $(\rho f)\ (\rho v)\ v' \wedge P\ v'$. The definition of AppEval then gives us $(\rho f)\,(\rho v) \Downarrow v'$, which is equivalent to $\rho(f\, v) \Downarrow v'$.

case $[\![\text{if } v \text{ then } t_1 \text{ else } t_2]\!]_\rho$: We have $[\![t_1]\!]_\rho P \implies \exists r.\, \rho t_1 \Downarrow r \wedge Pr$ and $[\![t_2]\!]_\rho P \implies \exists r.\, \rho t_2 \Downarrow r \wedge Pr$ by the induction hypothesis. Assume $[\![\text{if } v \text{ then } t_1 \text{ else } t_2]\!]_\rho P$. Then we know $(v = \mathit{true} \Rightarrow [\![t_1]\!]_\rho P) \wedge (v = \mathit{false} \Rightarrow [\![t_2]\!]_\rho P)$. For the case that $v = \mathit{true}$, we know by the induction hypothesis that $\exists r.\, \rho t_1 \Downarrow r \wedge Pr$. We can then apply rule IFT, which gives us $\exists r.\, \text{if } \mathit{true} \text{ then } \rho t_1 \text{ else } \rho t_2 \Downarrow r$, which is the same as $\exists r.\, \rho(\text{if } \mathit{true} \text{ then } t_1 \text{ else } t_2) \Downarrow r$ Equivalently for the case where $v = \mathit{false}$.

case $[\![\text{let } x = t_1 \text{ in } t_2]\!]_\rho$: Assume $[\![\text{let } x = t_1 \text{ in } t_2]\!]_\rho P$, which gives us $\exists P'.\, [\![t_1]\!]_\rho P' \wedge (\forall X.\, P'X \Rightarrow [\![t_2]\!]_{(\rho, x \mapsto X)} P)$. We then have to find an r such that $\rho(\text{let } x = t_1 \text{ in } t_2) \Downarrow r$ and $P\, r$. By the induction hypothesis we have that $\forall P.\, [\![t_1]\!]_\rho P \implies \exists r_1.\, \rho t_1 \Downarrow r_1 \wedge P\, r_1$. We can instantiate this P with P' and thus get $\exists r_1.\, \rho t_1 \Downarrow r_1 \wedge P' r_1$. We can now instantiate X in our assumption with this r_1, so we have $[\![t_2]\!]_{(\rho, x \mapsto r_1)}\, P$. We can then use the induction hypothesis, giving us $\exists r_2.\, [\rho, x \mapsto r_1] t_2 \Downarrow r_2 \wedge P\, r_2$. Since $x \notin \rho$, this is equivalent to stating $\exists r_2.\, [x \mapsto r_1] \rho t_2 \Downarrow r_2 \wedge P\, r_2$. From the semantics rule LET we can conclude that let $x = \rho t_1$ in $\rho t_2 \Downarrow r_2$ and thus $\rho(\text{let } x = t_1 \text{ in } t_2) \Downarrow r_2$.

case $[\![\text{let rec} f = \lambda x.\, t_1 \,\text{in}\, t_2]\!]_\rho$: The assumption is $\forall F.\, (\forall X.\, \forall P'.\, [\![t_1]\!]_{(\rho, f\mapsto F, x\mapsto X)} P' \Rightarrow \text{AppReturns}\, F\, X\, P') \;\Rightarrow\; [\![t_2]\!]_{(\rho, f\mapsto F)} P$. We first prove the part between the brackets, instantiating F with $(\mu f.\, \lambda x.\, \rho t_1)$. We take an arbitrary X (whose value we will denote with v_1) and P', giving us the assumption $[\![t_1]\!]_{(\rho, f\mapsto(\mu f.\lambda x.\rho t_1), x\mapsto v_1)} P'$ and goal $\text{AppReturns}\,(\mu f.\, \lambda x.\, \rho t_1)\, v_1\, P'$. We can use the induction hypothesis to get $\exists r_1.\, [\rho, f \mapsto (\mu f.\, \lambda x.\, \rho t_1), x \mapsto v_1] t_1 \Downarrow r_1 \wedge P'\, r_1$, which is equivalent to $\exists r_1.\, [f \mapsto (\mu f.\, \lambda x.\, \rho t_1), x \mapsto v_1] \rho t_1 \Downarrow r_1 \wedge P'\, r_1$.

From the semantics rule APP we get that $(\mu f = \lambda x.\, \rho t_1)\, v_1 \Downarrow r_1$. This gives us $\text{AppEval}\,(\mu f.\, \lambda x.\, \rho t_1)\, v_1\, r_1$ and thus $\text{AppReturns}\,(\mu f.\, \lambda x.\, \rho t_1)\, v_1\, P'$.

We now have that $[\![t_2]\!]_{(\rho, f\mapsto(\mu f.\lambda x.t_1))}\, P$. We use the induction hypothesis to get $\exists r_2.\, [\rho, f \mapsto (\mu f.\, \lambda x.\, t_1)] t_2 \Downarrow r_2 \wedge P\, r_2$, which is the same as $\exists r_2.\, [f \mapsto (\mu f.\, \lambda x.\, t_1)] \rho t_2 \Downarrow r_2 \wedge P\, r_2$. Using the semantics rule REC we then get $\exists r_2.\, (\text{let rec } f = \lambda x.\rho t_1 \text{ in } \rho t_2) \Downarrow r_2 \wedge P\, r_2$ and thus $\exists r_2.\, \rho(\text{let rec } f = \lambda x.t_1 \text{ in } t_2) \Downarrow r_2 \wedge P\, r_2$

Completeness is the converse of soundness. If soundness states that characteristic formulas describe their corresponding language, then completeness means that for every program that satisfies a set of postconditions, there is a characteristic formula that holds for exactly those postconditions. It states that for any term there is a corresponding characteristic formula that describes its behaviour. In other words, that any correct program can be proved correct using characteristic formulas. If a term t runs and returns some value v, then the characteristic formula of t will hold for the *most general specification* of t. In order to prove the completeness of CF, we will need a definition of this most general specification. Completeness can then be defined as, for a term t and a value v:

$$t \Downarrow v \implies [\![t]\!](\, \text{mgs}_t) \tag{2.1}$$

Here (mgs_t) denotes the most general specification of term t. For an ordinary v containing just a value, this most general specification is just $(= v)$, the partial application of equality on v, but if v contains a function this will not work.

2.6 Example of CF

2.6.1 Example program

The calculation of a CF will be demonstrated in detail on a simple example program double, which is a simple recursive program that doubles its input. It is by no means efficient but since it contains most of the language constructs it is a good example.

```
let rec double =
    λx.
        if (zero x) then 0
        else let y = double (pred_1 x) in
            suc_2 y
in
    double
```

It contains members of the families of functions pred_n and suc_n, which are unbound in this program. They apply a predecessor or successor function n times. It also contains the function zero, which checks if its argument is 0. These can be seen as library functions. In our example proof we will assume some specifications about them:

$$\forall x.\, \text{AppReturns pred}_n\ x\ (\lambda v.\, v = x - n)$$
$$\forall x.\, \text{AppReturns suc}_n\ x\ (\lambda v.\, v = x + n)$$
$$\forall x.\, \text{AppReturns zero}\ x\ (\lambda v.\, x = 0 \Rightarrow v = \mathit{true} \wedge x \neq 0 \Rightarrow v = \mathit{false})$$

We will also substitute them implicitly in the CF calculation.

2.6.2 CF Calculation

First the program needs to be reduced to the normal form described in section 3.2, where terms in if-conditions and function arguments are instead bound by a let-expression:

```
let rec double =
    $\lambda x.$
        let $a =$ (zero $x$) in
            if $a$ then 0
            else let $b =$ (pred$_1$ $x$) in
                let $y =$ double $b$ in
                    suc$_2$ $y$
in
    double
```

We then apply the CF function ($[\![ ]\!]$):

```
$[\![$let rec double =
    $\lambda x.$
        let $a =$ (zero $x$) in
            if $a$ then 0
            else let $b =$ (pred$_1$ $x$) in
                let $y =$ double $b$ in
                    suc$_2$ $y$
in
    double
$]\!]_\emptyset$
```

First the let rec is calculated:

```
λP. ∀D.
    (∀X. ∀P'.
        ⟦let $a =$ (zero $x$) in
        if $a$ then 0
        else let $b =$ (pred$_1$ $x$) in
            let $y =$ double $b$ in
                suc$_2$ $y$
        $⟧_{(\text{double} \mapsto D, x \mapsto X)} P'$
    $\Rightarrow$
        AppReturns $D\,x\,P'$
    )
$\Rightarrow$
    $⟦\text{double}⟧_{(\text{double} \mapsto D)} P$
```

Here P is a predicate of type $A \to$ Prop and double is of type Func.

We calculate the let a expression first:

```
⟦let $a =$ (zero $x$) in
    if $a$ then 0
    else let $b =$ (pred$_1$ $x$) in
        let $y =$ double $b$ in
            suc$_2$ $y⟧_{(\text{double} \mapsto D, x \mapsto X)} P'$
```

Which becomes:

$\exists P''.$

$$\llbracket(\text{zero } x)\rrbracket_{(\text{double}\mapsto D, x\mapsto X)} P''$$

$\wedge$

$(\forall A.$

$P'' A$

$\Rightarrow$

$$\begin{aligned}&\llbracket\text{if } a \text{ then } 0\\&\text{else let } b = (\text{pred}_1\ x) \text{ in}\\&\quad\text{let } y = \text{ double } b \text{ in}\\&\qquad\text{suc}_2\ y\rrbracket_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A)} P'\end{aligned}$$

$)$

First we calculate $\llbracket(\text{zero } x)\rrbracket_{(\text{double}\mapsto D, x\mapsto X)}\ P''$, which becomes AppReturns zero $X\ P''$. Then we calculate the CF for the if-statement:

$$\begin{aligned}&\llbracket\text{if } a \text{ then } 0\\&\text{else let } b = (\text{pred}_1\ x) \text{ in}\\&\quad\text{let } y = \text{ double } b \text{ in}\\&\qquad\text{suc}_2\ y\rrbracket_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A)} P'\end{aligned}$$

Which is equal to:

$$A = \text{ true } \Rightarrow P'\,0$$

$\wedge$

$$\begin{aligned}&A = \text{ false } \Rightarrow\\&\quad\llbracket\text{let } b = (\text{pred}_1\ x) \text{ in}\\&\qquad\text{let } y = \text{ double } b \text{ in}\\&\qquad\quad\text{suc}_2\ y\rrbracket_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A)} P'\end{aligned}$$

We also applied ($\llbracket\,\rrbracket$) on 0, which is a literal and just becomes the application of predicate P' on 0. Then we calculate the CF of the expression in the else-part:

$$
\begin{aligned}
&[\![\text{let } b = (\text{pred}_1\ x) \text{ in}\\
&\qquad \text{let } y = \ \text{double } b \text{ in}\\
&\qquad\qquad \text{suc}_2\ y]\!]_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A)} P'
\end{aligned}
$$

Which is the CF of two nested let-expressions. Since we have already seen how the CF of a let-expression is calculated we do the two nested expression in one step:

$$
\begin{aligned}
&\exists P''.\\
&\qquad [\![\text{pred}_1\ x]\!]_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A)} P''\\
&\wedge\\
&\qquad \forall B.\, P''\, B \Rightarrow\\
&\qquad\qquad \exists P'''.\\
&\qquad\qquad\qquad\qquad [\![\text{double } b]\!]_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A, b\mapsto B)} P'''\\
&\qquad\qquad\quad \wedge\\
&\qquad\qquad\qquad\qquad \forall Y.\, P'''\, Y \Rightarrow\\
&\qquad\qquad\qquad\qquad\qquad [\![\text{suc}_2\ y]\!]_{(\text{double}\mapsto D, x\mapsto X, a\mapsto A, b\mapsto B, y\mapsto Y)} P'
\end{aligned}
$$

The few remaining expressions are all function applications, which become AppReturns $\text{pred}_1\, X\, P''$, AppReturns $D\, B\, P'''$ and AppReturns $(+2)\, Y\, P'$.

Putting everything together, the full characteristic formula of this program becomes:

```
$\lambda P.\, \forall D.$
    $(\forall X.\, \forall P'.$
        $\exists P''.$
                $\text{AppReturns}\ \text{zero}\ X\ P''$
            $\wedge$
                $($
                    $\forall A.\, P''\, A$
                $\Rightarrow$
                        $A = \text{ true } \Rightarrow P'\ 0$
                    $\wedge$
                        $A = \text{ false } \Rightarrow$
                            $\exists P''.$
                                $\text{AppReturns}\ \text{pred}_1\ X\ P''$
                            $\wedge$
                                $\forall B.\, P''\ B \Rightarrow$
                                    $\exists P'''.$
                                            $\text{AppReturns}\ D\ B\ P'''$
                                        $\wedge$
                                            $\forall Y.\, P'''\ Y \Rightarrow$
                                                $\text{AppReturns}\ \text{suc}_2\ Y\ P'$
                $)$
    $\Rightarrow$
        $\text{AppReturns}\ D\ x\ P'$
    $)$
$\Rightarrow$
    $D\ P$
```

2.6.3 Example Proof

As a demonstration of the power of characteristic formulas in correctness proofs, we will show a proof of a specification for the program double, using the characteristic formula calculated in the previous section. An obvious specification for the program double would be:

$$\forall x.\, \forall n.\, n = 2 * x \implies (\text{double } x) \ = n$$

The characteristic formula is:

$$\begin{aligned}
&\lambda P.\, \forall D.\\
&\quad \forall X.\, \forall P'.\\
&\qquad [\![\text{let } a = (\text{zero } x) \text{ in}\\
&\qquad \text{if } a \text{ then } 0\\
&\qquad \text{else let } b = (\text{pred}_1\ x) \text{ in}\\
&\qquad\quad \text{let } y = \ \text{double } b \text{ in}\\
&\qquad\qquad \text{suc}_2\ y\\
&\qquad]\!]_{(\text{double} \mapsto D, x \mapsto X)} P'\\
&\quad \Rightarrow\\
&\qquad \text{AppReturns}\, D\, X\, P'\\
&\Rightarrow\\
&\quad P D
\end{aligned}$$

Also recall the assumptions we made about pred_n, suc_n and zero:

$$\begin{aligned}
&\forall x.\, \text{AppReturns pred}_n\ x\ (\lambda v.\, v = x - n)\\
&\forall x.\, \text{AppReturns suc}_n\ x\ (\lambda v.\, v = x + n)\\
&\forall x.\, \text{AppReturns zero}\ x\ (\lambda v.\, x = 0 \Rightarrow v = \mathit{true} \wedge x \neq 0 \Rightarrow v = \mathit{false})
\end{aligned}$$

We instantiate P as our specification: $P = (\lambda f.\, \forall x.\, \forall n.\, n = 2 * x \implies \text{AppReturns}\, f\, x\, (\lambda v.\, v = n))$ and D as $\overline{\text{double}}$. Note that double is a variable in the language, D (in the characteristic formula below) is a bound variable that quantifies over all closure values, and $\overline{\text{double}}$ is some function closure (this can be seen as the actual value that the variable double in the code is bound to, but is actually more general). This gives us the following proposition:

$$
\begin{array}{l}
\quad \forall X.\, \forall P'. \\
\qquad [\![\text{let } a = (\text{zero } x) \text{ in} \\
\qquad \text{if } a \text{ then } 0 \\
\qquad \text{else let } b = (\text{pred}_1\ x) \text{ in} \\
\qquad\qquad \text{let } y = \ \text{double } b \text{ in} \\
\qquad\qquad\quad \text{suc}_2\ y \\
\qquad]\!]_{(\text{double} \mapsto \overline{\text{double}}, x \mapsto X)} P' \\
\quad \Rightarrow \\
\qquad \text{AppReturns}\, \overline{\text{double}}\, X\, P' \\
\Rightarrow \\
\quad \forall x.\, \forall n.\, n = 2 * x \implies \text{AppReturns}\, \overline{\text{double}}\ x\ (= n)
\end{array}
$$

We prove this by induction over x. For the base case, we instantiate X as 0 and P' as $(\lambda v.\, v = 0)$. We have to prove the left side of the upper implication, which is:

$$
\begin{array}{l}
\exists P''. \\
\qquad \text{AppReturns zero}\, 0\, P'' \\
\quad \wedge \\
\qquad (\\
\qquad\quad \forall A.\, P'' A \\
\qquad \Rightarrow \\
\qquad\quad [\![\text{if } a \text{ then } 0 \\
\qquad\quad \text{else let } b = (\text{pred}_1\ x) \text{ in} \\
\qquad\qquad \text{let } y = \ \text{double } b \text{ in} \\
\qquad\qquad\quad \text{suc}_2\ y]\!]_{(\text{double} \mapsto \overline{\text{double}}, x \mapsto 0, a \mapsto A)} P' \\
\qquad)
\end{array}
$$

We instantiate P'' as $(\lambda v.\, v = \mathit{true})$, which proves the first part of the conjunction. We then have to prove the right side of the implication in the second part of the conjuction, which is:

$$
\begin{array}{l}
\qquad A = \text{ true } \Rightarrow (\lambda v.\, v = 0)\; 0 \\
\wedge \\
\qquad A = \text{ false } \Rightarrow \\
\qquad\quad [\![\text{let } b = (\text{pred}_1\ x) \text{ in} \\
\qquad\qquad \text{let } y = \text{ double } b \text{ in} \\
\qquad\qquad\quad \text{suc}_2\ y]\!]_{(\text{double}\mapsto\overline{\text{double}}, x\mapsto 0)} P'
\end{array}
$$

We have that $A =$ true, so both parts of the conjunction are trivially true. This concludes the proof of the case where $x = 0$.
For the inductive case we have that

$$n = 2 * x \implies \text{AppReturns}\ \overline{\text{double}}\ x\ (= n)$$

for some n and x. We have to prove:

$$n' = 2 * (x + 1) \implies \text{AppReturns}\ \overline{\text{double}}\ x + 1\ (= n')$$

for some n', which we will instantiate as $n + 2$.
We instantiate $\forall X$ with $x + 1$ and $\forall P'$ with $(\lambda v.\, v = n + 2)$ in the CF and we once again have to prove the left side of the implication:

$$
\begin{array}{l}
\exists P''. \\
\qquad \text{AppReturns zero}\,(x+1)\,P'' \\
\quad \wedge \\
\qquad (\\
\qquad\quad \forall A.\, P'' A \\
\qquad \Rightarrow \\
\qquad\quad [\![\text{if } a \text{ then } 0 \\
\qquad\quad \text{else let } b = (\text{pred}_1\ x) \text{ in} \\
\qquad\qquad \text{let } y = \text{ double } b \text{ in} \\
\qquad\qquad\quad \text{suc}_2\ y]\!]_{(\text{double}\mapsto\overline{\text{double}}, x\mapsto(x+1), a\mapsto A)}\ (\lambda v.\, v = n + 2) \\
\qquad)
\end{array}
$$

We instantiate P'' as $(\lambda v.\, v = \mathit{false})$, since we have that $x \geq 0$ and thus

$x + 1 \neq 0$. We then have to prove

$$\begin{array}{l} \quad A = \text{ true } \Rightarrow (\lambda v.\, v = n+2)\; 0 \\ \wedge \\ \quad A = \text{ false } \Rightarrow \\ \qquad [\![\text{let } b = (\text{pred}_1\; x) \text{ in} \\ \qquad\quad \text{let } y = \text{ double } b \text{ in} \\ \qquad\qquad \text{suc}_2\; y]\!]_{(\text{double} \mapsto \overline{\text{double}}, x \mapsto (x+1))}\; (\lambda v.\, v = n+2) \end{array}$$

We have that $A = \mathit{false}$ which makes the first part of the conjunction trivially true, so now we have to prove:

$$\begin{array}{l} \exists P''. \\ \quad \text{AppReturns}\,\text{pred}_1\,(x+1)\,P'' \\ \wedge \\ \quad \forall B.\, P''\,B \Rightarrow \\ \qquad \exists P'''. \\ \qquad\qquad \text{AppReturns}\;\overline{\text{double}}\;B\;P''' \\ \qquad\quad \wedge \\ \qquad\qquad \forall Y.\, P'''\,Y \Rightarrow \\ \qquad\qquad\quad \text{AppReturns suc}_2\,Y\;(\lambda v.\, v = n+2) \end{array}$$

Instantiating P'' with $(\lambda v.\, v = x)$ gives us AppReturns $\text{pred}_1\,(x+1)\,(\lambda v.\, v = x)$, which is true. In the second part of the conjunction, we have to prove the right side of implication. We instantiate P''' as $(\lambda v.\, v = n)$. We have $(\lambda v.\, v = x)\,B$ so by application of the induction hypothesis we have AppReturns $\overline{\text{double}}$ B $(\lambda v.\, v = n)$. Then we have AppReturns suc_2 Y $(\lambda v.\, v = n+2)$, since $Y = n$. We now have that AppReturns $\overline{\text{double}}$ $(x+1)$ $(\lambda v.\, v = n+2)$, proving the inductive case.

2.7 Example in CakeML

Our example program is easily translated to actual CakeML code:

```
fun double x =
if (x = 0)
  then 0
  else (let val y = (double (x - 1)); in (y + 2) end);
```

Then we set up our proofscript. We need the following libraries:

```
open preamble
open ml_translatorTheory cfTacticsBaseLib cfTacticsLib
local open ml_progLib basisProgTheory in end
```

The library `ml_translatorTheory` contains various predicates to describe relations between HOL logic values and values in the deep embedding of CakeML. These predicates will be used in specifications. The libraries `cfTacticsBaseLib` and `cfTacticsLib` contain the various tactics used in CF proofs. `ml_progLib` is used for managing the state after evaluation of toplevel declarations (such as the declaration of our `double` function). We fetch a basic state from basisProgTheory containing basic defintions and comes with specifications for these definitions:

```
val basis_st =
  ml_progLib.unpack_ml_prog_state
    basisProgTheory.basis_prog_state
```

We can now define our function:

```
val double_def = process_topdecs
  `fun double x =
    (if (x = 0)
      then 0
      else (let val y = (double (x - 1)); in (y + 2) end));`
```

This definition is then added to the basis state:

```
val st = ml_progLib.add_prog double_def ml_progLib.pick_name basis_st
```

Now we can start proving a specification for `double`.

```
val double_spec = store_thm ("double_spec",
```

Specifications for CF proofs should be of a certain form, as explained in `cakeml/characteristic/examples/cf_tutorialScript.sml`, found in the CakeML repository:

```
!x1..xn argv1.. argvm.
 facts_about_xi_argvj x1 .. xn .. argv1 .. argvm ==>
 app (p:'ffi ffi_proj) ^(fetch_v "name" st) [argv1, argv2,...]
   precondition postcondition
```

where

- `name` is the name of the function in the toplevel declaration we added to the `ml_prog_state`
- `st` is the state containing the function
- `x1`, ..., `xn` are values from the logic (HOL-values) used in the specification
- `argv1`, ..., `argvm` are the arguments of the function, as values of the deep embedding (cakeML values as represented in HOL4 logic)
- `facts_about_xi_argvj` contain predicates from `ml_translatorTheory` that relate values of the logic (`xi`) to values of the deep embedding (`argvj`)
- `precondition` is a heap predicate (of type `hprop`), and describes the state of the memory heap before execution of the function. The syntax for heap predicates we will use includes:
  - `emp`, the empty heap
  - `& P`, where `&` is an operation that lifts a boolean value `P: bool` to the level of heap predicates
  -
- `postcondition` is a heap predicate which describes the state after execution of the function. A function can either return a value or raise an exception, there are helper functions that can deal with either case:
  - `POSTv v. Q` is a post-condition that asserts that execution of the function always returns a value, which in this case is represented by `v`, and that the heap satisfies heap predicate `Q`. Usually Q also specifies what value `v` should have.
  - `POSTe v. Q` works much the same, but asserts that execution of the function will always raise an exception.
  - `POST Qv Qe` is the general case. `Qv` is a post-condition where the function returns a value, and `Qe` is a post-condition where the function raises an exception.
- the exclamation mark `!` represents a universal quantifier ($\forall$)

Now that we know what CF-specifications should look like, we can define a specification for double:

```
!x xv.
 NUM x xv ==>
 app (p:'ffi ffi_proj) ^(fetch_v "double" st) [xv]
   emp (POSTv v. &NUM (2 * x) v)
```

This specification states that, given a HOL-value x and a value of the deep embedding xv, if NUM x xv, where NUM is a predicate from ml_translatorTheory which states that xv represents a natural number x in the deep embedding, the application of the top level declaration double in the state st on xv will satisfy the precondition emp (since it is a pure function) and postcondition (POSTv v. &NUM (2 * x) v) which states that the value which the application of double reduces to will be a representation of the number 2 * x in the deep embedding. In this specification, the predicate app fulfills the same role as the predicate AppReturns in the formal definitions of this chapter, and the specification of double in the pen-and-paper proof. The specification in HOL is in essence the same as the specification in the previous section, but in the pen-and-paper proof we ignored that the program logic and proof logic were different, and in HOL these layers are made explicit.

The mechanized proof of this specification is very similar to the pen-and-paper proof. We will use induction on the argument, x:

```
Induct_on ‘x‘
```

which results in the following sequents:

```
 !xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
------------------------------------
!xv.
    NUM (SUC x) xv  app p double_v [xv] emp (POSTv v. &NUM (2 * SUC x) v)



!xv. NUM 0 xv  app p double_v [xv] emp (POSTv v. &NUM (2 * 0) v)
```

We then calculate the CF of double in state st:

```
xcf "double" st
```

and start proving the base case:

```
 NUM 0 xv
------------------------------------
cf_let (SOME "a")
  (cf_app p (Var (Short "=")) [Var (Short "x"); Lit (IntLit 0)])
```

```
(cf_if (Var (Short "a")) (cf_lit (IntLit 0))
   (cf_let (SOME "b")
      (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
      (cf_let (SOME "y")
         (cf_app p (Var (Short "double")) [Var (Short "b")])
         (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)]))))
(...) emp (POSTv v. &NUM 0 v)
```

The CF looks different from the CF calculated by hand, because the program is first reduced to a 'normal form', in which if-statement conditions get replaced by a variable bound by a let-expression which binds this variable to the expression that was in the condition. In our pen-and-paper proof, we also treated operators on numbers, such as $+$, $-$ and $=$, more implicitly for simplification, but here they become function applications. Since the top-level expression is a let-expression, we use the tactic `xlet`. As in our paper proof, we have to provide an postcondition when proving statements about a characteristic formula of a let-expression:

```
xlet ‘POSTv bv. &BOOL T bv‘
```

In this case, since we have `NUM 0 xv`, which means that x is 0 in the deep embedding, and the condition expression is `x = 0`, we instantiate the post-condition of the let-expression with `POSTv bv. &BOOL T bv` which states that the bound expression reduces to 'True' in the deep embedding.

```
 0.  NUM 0 xv
  1.  BOOL T bv
------------------------------------
cf_if (Var (Short "a")) (cf_lit (IntLit 0))
  (cf_let (SOME "b")
     (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
     (cf_let (SOME "y")
        (cf_app p (Var (Short "double")) [Var (Short "b")])
        (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)])))
  (...) emp (POSTv v. &NUM 0 v)


  NUM 0 xv
------------------------------------
cf_app p (Var (Short "=")) [Var (Short "x"); Lit (IntLit 0)] (...) emp
  (POSTv bv. &BOOL T bv)
```

Now we have to prove that, given this postcondition, the program is correct, but also that the postcondition holds for the bound expression `x = 0`. The latter will be proved first, the second sequent describes this case. It states that the application of the function `=` on arguments `x` and `0` will satisfy the precondition `emp` and the postcondition described before. The tactic used to deal with a CF of an application expression does not specify types, so we will need to prove an extra theorem for type specivity:

```
    val eq_v_NUM_thm = Q.prove(‘(NUM --> NUM --> BOOL) $= eq_v‘,
    metis_tac[DISCH_ALL mlbasicsProgTheory.eq_v_thm,EqualityType_NUM_BOOL]);
```

We then use the more specific tactic `xapp_spec` along with this theorem:

```
    xapp_spec eq_v_NUM_thm
```

which results in:

```
    NUM 0 xv
   ------------------------------------
   ?H2 x1 x0.
       (NUM x0 xv /\ NUM x1 (Litv (IntLit 0))) /\ emp ==>> emp * H2 /\
       (POSTv v. &BOOL (x0 = x1) v) *+ H2 ==+> (POSTv bv. &BOOL T bv) *+ GC
```

and can be simplified to:

```
    xsimpl

    NUM 0 xv
   ------------------------------------
   ?x0. NUM x0 xv /\ !v. BOOL (x0 = 0) v  BOOL T v
```

Where `?` represents an existential quantifier (∃). We instantiate `x0` with `0` and use a rewrite tactic `fs[]` to prove this subgoal:

```
    qexists_tac ‘0‘

    NUM 0 xv
   ------------------------------------
   NUM 0 xv /\ !v. BOOL (0 = 0) v  BOOL T v

    fs []

    Goal proved.
    ...
    Remaining subgoals:
val it =



      0.  NUM 0 xv
     1.  BOOL T bv
   ------------------------------------
   cf_if (Var (Short "a")) (cf_lit (IntLit 0))
```

```
 (cf_let (SOME "b")
    (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
    (cf_let (SOME "y")
       (cf_app p (Var (Short "double")) [Var (Short "b")])
       (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)])))
 (...) emp (POSTv v. &NUM 0 v)
```

We use the tactic `xif` for dealing with a CF of an if-expression:

```
xif

b.
   BOOL b bv  (b  cf_lit (IntLit 0) (...) emp (POSTv v. &NUM 0 v))
   (¬b
    cf_let (SOME "b")
      (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
      (cf_let (SOME "y")
         (cf_app p (Var (Short "double")) [Var (Short "b")])
         (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)]))
      (...) emp (POSTv v. &NUM 0 v))
```

We can instantiate `b` with `T`, and simplify

```
qexists_tac ‘T‘ \\ xsimpl

0.  NUM 0 xv
 1.  BOOL T bv
------------------------------------
cf_lit (IntLit 0) (...) emp (POSTv v. &NUM 0 v)
```

We then finish the proof by using the CF tactic for dealing with literals, and simplifying:

```
    xlit \\ xsimpl

    Goal proved.
    ...
    Remaining subgoals:
val it =
    xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
   ------------------------------------
   xv.
       NUM (SUC x) xv  app p double_v [xv] emp (POSTv v. &NUM (2 * SUC x) v)
```

This concludes the base case, where the argument of `double` equals 0. We now move on to the inductive case. Once again, we calculate the CF of `double`:

```
    xcf "double" st

    0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
   ------------------------------------
   cf_let (SOME "a")
     (cf_app p (Var (Short "=")) [Var (Short "x"); Lit (IntLit 0)])
     (cf_if (Var (Short "a")) (cf_lit (IntLit 0))
        (cf_let (SOME "b")
           (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
           (cf_let (SOME "y")
              (cf_app p (Var (Short "double")) [Var (Short "b")])
              (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)]))))
     (...) emp (POSTv v. &NUM (2 * SUC x) v)
```

Once again we have to deal with the CF for let-expression arising from the if-statement. This is done in mostly the same way as before, only this time with the result being T:

```
    xlet ‘POSTv bv. &BOOL F bv‘

     0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
   ------------------------------------
   cf_if (Var (Short "a")) (cf_lit (IntLit 0))
     (cf_let (SOME "b")
        (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
        (cf_let (SOME "y")
           (cf_app p (Var (Short "double")) [Var (Short "b")])
           (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)])))
     (...) emp (POSTv v. &NUM (2 * SUC x) v)


      0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
   ------------------------------------
   cf_app p (Var (Short "=")) [Var (Short "x"); Lit (IntLit 0)] (...) emp
     (POSTv bv. &BOOL F bv)
```

The bottom subgoal is practically the same as in the previous if-expression, and proven in the same way:

```
    xapp_spec eq_v_NUM_thm \\
    xsimpl \\
    qexists_tac ‘SUC x‘ \\
fs []
```

The only difference is we instantiate the x0 with (SUC x), since that is the value of our argument in the inductive case, and because it is a succesor of some x (which is a natural number) it cannot be equal to 0.

```
    Goal proved.
 [..]
 cf_app p (Var (Short "=")) [Var (Short "x"); Lit (IntLit 0)] (...) emp
    (POSTv bv. &BOOL F bv)


Remaining subgoals:
val it =
      0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
   ------------------------------------
   cf_if (Var (Short "a")) (cf_lit (IntLit 0))
     (cf_let (SOME "b")
        (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
        (cf_let (SOME "y")
           (cf_app p (Var (Short "double")) [Var (Short "b")])
           (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)])))
     (...) emp (POSTv v. &NUM (2 * SUC x) v)
```

Once again we use the tactic xif, this time instantiating the condition boolean with F (False).

```
    xif \\
    qexists_tac ‘F‘ \\
    xsimpl
```

```
    0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
   ------------------------------------
   cf_let (SOME "b")
     (cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)])
     (cf_let (SOME "y") (cf_app p (Var (Short "double")) [Var (Short "b")])
        (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)])) (...)
     emp (POSTv v. &NUM (2 * SUC x) v)
```

Another change to the program that happened in normalization is that arguments of a function that are an expression and not a variable get bound to a variable by a preceding let-expression. In this case, the expression `x - 1` is bound to the variable `b`. This expression should reduce to the value of the argument minus one, in this case `(SUC x) - 1`. We use the `xlet`-tactic as such:

```
 xlet ‘POSTv cv. &NUM ((SUC x) - 1) cv‘

 0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
  1.  NUM (SUC x) xv
  2.  BOOL F bv
  3.  NUM x cv
------------------------------------
cf_let (SOME "y") (cf_app p (Var (Short "double")) [Var (Short "b")])
  (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)]) (...) emp
  (POSTv v. &NUM (2 * SUC x) v)


   0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
  1.  NUM (SUC x) xv
  2.  BOOL F bv
------------------------------------
cf_app p (Var (Short "-")) [Var (Short "x"); Lit (IntLit 1)] (...) emp
  (POSTv cv. &NUM (SUC x  1) cv)
```

Starting with the bottom sequent we use `xapp` and simplify:

```
 xapp // xsimpl

  0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
  1.  NUM (SUC x) xv
  2.  BOOL F bv
------------------------------------
x0. INT x0 xv  v. INT (x0  1) v  NUM x v
```

We have a mix of `INT` and `NUM` terms, so we rewrite the `NUM` terms, since `NUM x xv` is equivalent to `INT (&SUC x) xv`.

```
 fs [NUM_def]

 0.  xv’.
          INT (&x) xv’
          app p double_v [xv’] emp (POSTv v. &INT (&(2 * x)) v)
  1.  INT (&SUC x) xv
  2.  BOOL F bv
------------------------------------
x0. INT x0 xv  v. INT (x0  1) v  INT (&x) v
```

Now it is clear that we have to instantiate `x0` with `&SUC x`, where `& : num -> int` denotes the injection function from natural numbers to integers, and simplify.

```
  qexists_tac ‘&SUC x‘ \\ xsimpl

   0.  xv’.
           INT (&x) xv’
           app p double_v [xv’] emp (POSTv v. &INT (&(2 * x)) v)
   1.  INT (&SUC x) xv
   2.  BOOL F bv
 ------------------------------------
 v. INT (&SUC x  1) v  INT (&x)
```

To prove this we need to prove an extra theorem:

```
val eq_suc_sub1 = Q.prove(
‘!x. ( int_of_num (SUC x)) - (int_of_num 1) = int_of_num x‘,
intLib.ARITH_TAC);
```

where `int_of_num` is the type-specific version of `&` (`&` is also the injection function for rational numbers and real numbers). `intLib.ARITH_TAC` is a decision procedure that deals with linear arithmetic over integers and natural numbers. We can now use this theorem to prove our subgoal:

```
    fs [eq_suc_sub1]

    Goal proved.
    ...
    Remaining subgoals:
val it =


      0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
     3.  NUM x cv
   ------------------------------------
   cf_let (SOME "y") (cf_app p (Var (Short "double")) [Var (Short "b")])
     (cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)]) (...) emp
     (POSTv v. &NUM (2 * SUC x) v)
```

The last part of the proof is of a very similar structure as the previous subgoals. Because we had `&SUC x - 1` as postcondition of the previous let-expression, we should instantiate this one with `2 * x`, since `2 * (&SUC x - 1)` is `2 * x`.

```
    0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
     3.  NUM x cv
     4.  NUM (2 * x) nv
   ------------------------------------
   cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)] (...) emp
     (POSTv v. &NUM (2 * SUC x) v)


      0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
     3.  NUM x cv
   ------------------------------------
   cf_app p (Var (Short "double")) [Var (Short "b")] (...) emp
     (POSTv nv. &NUM (2 * x) nv)
```

The bottom subgoal can be proven easily using assumption 0:

```
    xapp // fs []

    Using a CF specification from the assumptions

Goal proved.
...
Remaining subgoals:
val it =


      0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
     3.  NUM x cv
     4.  NUM (2 * x) nv
   ------------------------------------
   cf_app p (Var (Short "+")) [Var (Short "y"); Lit (IntLit 2)] (...) emp
     (POSTv v. &NUM (2 * SUC x) v)
```

We use xapp and simplify:

```
    xapp \\ xsimpl

    0.  xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v)
     1.  NUM (SUC x) xv
     2.  BOOL F bv
     3.  NUM x cv
     4.  NUM (2 * x) nv
   ------------------------------------
   x0. INT x0 nv  v. INT (x0 + 2) v  NUM (2 * SUC x) v
```

Again we rewrite the `NUM` terms to `INT` terms:

```
    0.  xv'.
             INT (&x) xv'
             app p double_v [xv'] emp (POSTv v. &INT (&(2 * x)) v)
     1.  INT (&SUC x) xv
     2.  BOOL F bv
     3.  INT (&x) cv
     4.  INT (&(2 * x)) nv
   ------------------------------------
   x0. INT x0 nv  v. INT (x0 + 2) v  INT (&(2 * SUC x)) v
```

We instantiate `x0` with `&(2 * x)`, and simplify:

```
    qexists_tac `&(2 * x)` \\ xsimpl

    0.  xv'.
             INT (&x) xv'
             app p double_v [xv'] emp (POSTv v. &INT (&(2 * x)) v)
     1.  INT (&SUC x) xv
     2.  BOOL F bv
     3.  INT (&x) cv
     4.  INT (&(2 * x)) nv
   ------------------------------------
   v. INT (&(2 * x) + 2) v  INT (&(2 * SUC x)) v
```

We prove a similar theorem to `eq_suc_sub1`:

```
  val eq_suc_mul2 = Q.prove (
  `!x. ( int_of_num (2 * x)) + (int_of_num 2) = int_of_num (2 * SUC x)`,
  intLib.ARITH_TAC);
```

and use this to finalize the proof:

```
    fs [eq_suc_mul2]

    Goal proved.
    ...
val it =
   Initial goal proved.
    x xv. NUM x xv  app p double_v [xv] emp (POSTv v. &NUM (2 * x) v):
```

Chapter 3

Conclusions

Characteristic formulae are a relatively new and underutilized method of program verification using mechanized proofs. The most significant contribution of CF is that the automatic generation of the formulas removes the previously error-prone step of manually translating code to some embedding in the logic of the theorem prover. CF proofs are also relatively straightforward; someone who has experience with HOL4 should not have too hard of a time utilizing this technique.

This thesis examines both the theory and practise of using characteristic formulae for verification. It aims to clearly show the parallels between the formal definitions and the machinations within the theorem prover HOL4. This work can also be used as a somewhat gentle introduction to using characteristic formulae for program verification, demonstrating all the necessary steps to get one started on this task.

What has been examined in this thesis is only a minor part of what characteristic formulas can do, as the theoretical framework in this thesis has been significantly simplified. Characteristic formulas also exist for imperative programs, defining preconditions and postconditions for the program state before and after the program, using higher-order separation logic. I have chosen not to discuss this in the thesis due to time constraints.

How characteristic formulae compare to other methods of program verification is up to reader's judgement. It is certainly a very precise and efficient method, but it does somewhat restrict the usable proof tactics to the tactics specific to the characteristic formulae framework. Documentation in the case of CakeML is also slightly sparse or lacking, especially to someone with little to no experience using HOL4. We hope this thesis can provide a new user with a smoother start to using the technique.

Bibliography

[1] HOL Interactive Theorem Prover. `https://hol-theorem-prover.org/`.

[2] HOL Zero. `http://www.proof-technologies.com/holzero/`.

[3] Isabelle. `https://isabelle.in.tum.de`.

[4] ProofPower. `http://www.lemma-one.com/ProofPower/getting/`.

[5] The HOL Light theorem prover. `https://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[6] A modal characterization of observational congruence on finite terms of ccs. *Information and Control*, 68(1-3):125–145, 1986.

[7] Luca Aceto and Anna Ingolfsdottir. Characteristic Formulae: From Automata to Logic. *BRICS*.

[8] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris Diderot, 2010. `http://www.chargueraud.org/research/2010/thesis/thesis_final.pdf`.

[9] Michael J.C. Gordon. From lcf to hol: a short history. In Plotkin, G, Stirling, Colin P., and Tofte, Mads, editors, *Proof, Language, and Interaction*. MIT Press, 2000.

[10] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for cakeml. *Programming Languages and Systems*, pages 584–610, 2017.

[11] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and con- currency. *J. ACM*, 32(1):137–161, 1985.

[12] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. *Principles of Programming Languages (POPL)*, 2014.

[13] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[14] Magnus O. Myreen. Guide to hol4 interaction and basic proofs. `https://hol-theorem-prover.org/HOL-interaction.pdf`.

[15] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ml. *Journal of Functional Programming*, 24(2-3):284–315, 2014.

[16] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In *ESOP*, 2016.

[17] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for cakeml. *ICFP 2016*, 2016.

[18] CakeML Team. Cakeml. `https://github.com/CakeML/cakeml`, 2017.

Appendix A

Proofscript double

```
(*
  Example program double for thesis on characteristic formulae
*)

open preamble
open ml_translatorTheory cfTacticsBaseLib cfTacticsLib
local open ml_progLib basisProgTheory in end
open arithmeticTheory
open integerTheory

val _ = new_theory"double";

val pick_name = ml_progLib.pick_name;

val basis_st =
  ml_progLib.unpack_ml_prog_state
    basisProgTheory.basis_prog_state

val double_def = process_topdecs
  `fun double x =
    (if (x = 0)
      then 0
      else (let val y = (double (x - 1)); in (y + 2) end));`

val st = ml_progLib.add_prog double_def ml_progLib.pick_name basis_st

val eq_v_NUM_thm = Q.prove(`(NUM --> NUM --> BOOL) $= eq_v`,
  metis_tac[DISCH_ALL mlbasicsProgTheory.eq_v_thm,EqualityType_NUM_BOOL]);

val eq_suc_sub1 = Q.prove(
  `!x. ( int_of_num (SUC x)) - (int_of_num 1) = int_of_num x`,
  intLib.ARITH_TAC);

val eq_suc_mul2 = Q.prove (
  `!x. ( int_of_num (2 * x)) + (int_of_num 2) = int_of_num (2 * SUC x)`,
  intLib.ARITH_TAC);

VAL DOUBLE_SPEC = store_thm ("double_spec",
  ``!x xv.
     NUM x xv ==>
     app (p:'ffi ffi_proj) ^(fetch_v "double" st) [xv]
       emp (POSTv v. &NUM (2 * x) v)``,
  Induct_on `x` \\
  xcf "double" st
  THEN1  (xlet `POSTv bv. &BOOL T bv`
          THEN1 (xapp_spec eq_v_NUM_thm \\
            xsimpl \\ qexists_tac `0` \\
 fs [] ) \\
  xif \\ qexists_tac `T` \\ xsimpl \\ xlit \\ xsimpl) \\
    xcf "double" st
    xlet `POSTv bv. &BOOL F bv`
    THEN1 (xapp_spec eq_v_NUM_thm \\
        xsimpl \\
        qexists_tac `SUC x` \\
   fs []) \\
    xif \\
    qexists_tac `F` \\
    xsimpl \\
    xlet `POSTv cv. &NUM ((SUC x) - 1) cv`
    THEN1 (xapp \\
        xsimpl \\
        qexists_tac `&SUC x` \\
   fs [NUM_def]
        fs [eq_suc_sub1]) \\

    xlet `POSTv nv. &NUM (2 * x) nv`
    THEN1 (xapp \\
        fs []) \\
    xapp \\
    xsimpl \\
    qexists_tac `&(2 * x)` \\
    fs [eq_suc_mul2]);
```

Appendix B

Proofscript rev

In the original plan there was going to be a case study on programming and verifying the Unix program `rev` in CakeML, demonstrating the imperative and IO features of the CakeML implementation of characteristic formulas. Howver in the process the focus shifted towards the theoretical framework and `rev` was too complicated to verify by hand. Eventually we decided that the elaborated example of the double function was sufficient to demonstrate the basic idea of characteristic formulas, and due to time constraints `rev` was abandoned, but the (unfinished) proofscript is attached here as an appendix.

```
(*
  Simple rev program
*)
open preamble basis
val _ = new_theory"rev";
val _ = translation_extends"basisProg";
val rev_lines_def = Define‘
  rev_lines lines = (MAP (implode o REVERSE o explode) lines)‘;
val res = translate rev_lines_def
(*
val Prog_thm =
  get_ml_prog_state ()
  |> ml_progLib.clean_state
  |> ml_progLib.remove_snocs
  |> ml_progLib.get_thm
  |> REWRITE_RULE [ml_progTheory.ML_code_def];
*)
val rev = process_topdecs‘
  fun rev u =
    case TextIO.inputLinesFrom (List.hd (CommandLine.arguments()))
    of SOME lines =>
      (TextIO.print (concat (rev_lines lines));
      TextIO.output1 TextIO.stdOut #"\n")`;
val _ = append_prog rev;
val rev_spec = Q.store_thm("rev_spec",
  ` hasFreeFD fs  FILENAME fnm fnv
    cl = [pname; fname]
    contents = THE (ALOOKUP fs.files (File fname))
    app (p:'ffi ffi_proj) ^(fetch_v "rev" (get_ml_prog_state())) [fnv]
      (STDIO fs * COMMANDLINE cl)
      (POSTv uv. &UNIT_TYPE () uv *
              STDIO (add_stdout fs
   (concat [concat (rev_lines (MAP implode (splitlines contents)));
            strlit "\n"]))
         * COMMANDLINE cl)`
  simp [concat_def] \\
  strip_tac \\
  xcf "rev" (get_ml_prog_state()) \\
  xlet_auto >- (xcon \\ xsimpl) \\
  xlet_auto >- (xsimpl) \\
  xlet_auto >- (xsimpl) \\
  reverse(Cases_on`wfcl cl`) >- (rfs[COMMANDLINE_def] \\ xpull) \\
  reverse(Cases_on`STD_streams fs`) >- (fs[STDIO_def] \\ xpull) \\
  xlet_auto_spec(SOME inputLinesFrom_spec) >- (
xsimpl \\
    rfs[wfcl_def,validArg_def,EVERY_MEM] ) \\
  xmatch \\ fs[OPTION_TYPE_def] \\
  reverse conj_tac >- (EVAL_TAC \\ fs[]) \\ (* unfinished *)
```