INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Proving Properties of Programs Using Automatically Generated Models

By

Franck J.L. Binard

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillement of the requirements for the degree of

Master of Computer Science

In

Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering

Faculty of Engineering
University of Ottawa

May 2002
Ottawa, Canada

Abstract

This thesis describes a method for automatically generating theories in a first order logic with arithmetic for the purpose of proving properties about the programs. We build the logic and define the syntactic model generation algorithm. The models can be used to prove a substantial class of properties that are true of the programs they are generated from. Here, a model is a set of formulas of the logic that are interpreted as mathematical function definitions. These functions describe the computation steps of the algorithm.

Acknowledgments

This thesis wouldn't have been written without the moral and financial support of my supervisor, Dr. Amy Felty. The help that she supplied, both in the development of the theory and in the editing of this thesis can not be measured. I also couldn't have done this without the moral support and the (sometimes misguided) confidence of Vanja Kljajevic.

I also want to thank Hossein Yadollahi, for his friendship, and Dr. Odifreddi, for giving so much to Recursion Theory.

Contents

Chapter 1

Introduction

1.1 Problem Statement

The problem that is addressed in this thesis is the development of an efficient algorithmic way to translate programs written in a C like language into a syntactic model described by terms of a deductive logical system. Using the deductive calculus of the logical system, and the set of terms that is constructed, we show how properties about the code segments can be proven. It is then demonstrated how this method facilitates the automation of correctness proofs for a defined set of programs.

Remark 1.1.1 Our use of the term "model" in the context of this thesis is not to be confused with the mathematical logic notion of a model.

1.2 Approach and Motivation

1.2.1 Current Approach

A typical approach for producing direct correctness proofs is to use the Hoare Triple Logic (TLC) ([8], and a complete description of the Hoare Triple Logic system can be also found in [17]) format for program correctness. Because it is built along the idea that a program has a goal (called a *postcondition* or a *correctness property*), a Hoare Triple correctness proof implicitly assumes

that a code segment is meant to be doing something that is specific and is well defined. The proof then consists in using a set of rules that are applied to the structure of the program in order to prove that the computation that the program is meant to perform has indeed been done correctly by the time the program has gone through any of its possible execution paths, subject to satisfied precondition statements. Should either the preconditions or the postconditions be revised or added to, a new proof needs to be derived. The format of a Hoare Triple can be generalized to $|\phi|PrgStruct|\varphi|$, where ϕ and φ are well-formed expressions (usually in some first order logic) and *PrgStruct* is a programming construct (in a C type language). A Hoare Triple can roughly be interpreted as the following : if the statement ϕ can be proven to hold before the execution of *PrgStruct* (precondition), then the statement φ can be proven to hold immediatly after the execution of *PrgStruct* (postcondition).

A proof can then be viewed as a recursive decomposition of *PrgStruct* into $PrgStruct_1$, $PrgStruct_2$, $\cdots$ $PrgStruct_n$, following the structure of the programming language and finding a sequence
$|\phi_1|PrgStruct_1|\varphi_1|$, $|\phi_2|PrgStruct_2|\varphi_2|$, $\cdots$, $|\phi_n|PrgStruct_n|\varphi_n|$
such that each triple is provable using the Hoare Triple Rules as given below:

- $$\frac{(|\phi|)\ \text{FUNC1}\ (|\eta|) \qquad (|\eta|)\ \text{FUNC2}\ (|\psi|)}{(|\phi|)\ \text{FUNC1 ; FUNC2}\ (|\psi|)}\ \text{COMPOSITION}$$

- $$\frac{}{(|\phi[E/x_i]|)x_i = E;(|\phi|)}\ \text{ASSIGNMENT}$$

- $$\frac{(|\phi \wedge \beta|)\ \text{FUNC1}\ (|\psi|) \qquad (|\phi \wedge \neg\beta|)\ \text{FUNC2}\ (|\psi|)}{(|\phi|)\ \text{if}\ (\beta)\ \text{FUNC1 else FUNC2}\ (|\psi|)}\ \text{IF-STATEMENT}$$

- $$\frac{(|\psi \wedge \beta|)\ \text{FUNC1}\ (|\psi|)}{(|\psi|)\ \text{while}(\beta)\ \text{FUNC1}\ (|\psi \wedge \neg\beta|)}\ \text{PARTIAL WHILE}$$

- $$\frac{\phi' \longrightarrow \phi \qquad (|\phi|)\ \text{FUNC1}(|\psi|) \qquad \psi \longrightarrow \psi'}{(|\phi'|)\ \text{FUNC1}\ (|\psi'|)}\ \text{IMPLIED}$$

These rules can be found in [17].
Hoare Logic then provides for correctness proofs by creating a bridge from a programming language's syntax to a deductive logical system. The final product is a proof that the program satisfies the set postcondition. In chapter 3. Hoare Triple Logic is formally reintroduced. and used to prove some results that are of interest to this work.

1.2.2 Intrinsic Problems in Traditional Correctness Proof Techniques

The following example emphasizes the difficulty of automating the Hoare Proof Calculus method to produce correctness proofs.
Consider the following C language code segment :

```
void func (int x, int n){

            int a = 0;
            int b = 1;
            int c = 1;

            while(n > a){
                    a = a + 1;
                    b = x * b;
                    c = c * a;
            }
}
```

Assuming that at the moment the variable n passed on to the function has a value greater or equal to 0, Hoare triple logic allows for a proof that $b = x^n$ at the end of the function's execution. One such proof is given below.

$\lvert n \geq 0 \rvert$	PRE CONDITION

```
void func (int x, int n){
```

$\lvert n \geq 0 \rvert$	COPY
$\lvert (n \geq 0) \wedge (x^0 = 1) \rvert$	IMPLIED

```
int a = 0;
```

$\lvert (n \geq a) \wedge (x^a = 1) \rvert$	ASSIGNMENT

```
int b = 1;
```

$\lvert (n \geq a) \wedge (x^a = b) \rvert$	ASSIGNMENT

```
int c = 1;
```

$\lvert (n \geq a) \wedge (x^a = b) \rvert$	ASSIGNMENT

```
while(n > a){
```

$\lvert (n > a) \wedge (n \geq a) \wedge (x^a = b) \rvert$	HYP and INV
$\lvert (n > a) \wedge (x^a = b) \rvert$	IMPLIED
$\lvert (n \geq a + 1) \wedge (x^{a+1} = x * b) \rvert$	IMPLIED

```
a = a + 1;
```

$\lvert (n \geq a) \wedge (x^a = x * b) \rvert$	ASSIGNMENT

```
b = x * b;
```

$\lvert (n \geq a) \wedge (x^a = b) \rvert$	ASSIGNMENT

```
c = c * a;
```

$\lvert (n \geq a) \wedge (x^a = b) \rvert$	ASSIGNMENT

```
}
```

$\lvert\neg(n > a) \wedge (n \geq a) \wedge (x^a = b)\rvert$	PARTIAL WHILE
$\lvert(n = a) \wedge (x^a = b)\rvert$	IMPLIED
$\lvert x^n = b\rvert$	IMPLIED

```
}
```

$\lvert x^n = b\rvert$	POST CONDTION

Consider now the following proof on the same program :

$\lvert n \geq 0\rvert$	PRE CONDITION

```
void func (int x, int n){
```

$\lvert n \geq 0\rvert$	COPY
$\lvert(n \geq 0) \wedge (0! = 1)\rvert$	IMPLIED

```
int a = 0;
```

$\lvert(n \geq a) \wedge (a! = 1)\rvert$	ASSIGNMENT

```
int b = 1;
```

$\lvert(n \geq a) \wedge (a! = 1)\rvert$	ASSIGNMENT

```
int c = 1;
```

$\lvert(n \geq a) \wedge (a! = c)\rvert$	ASSIGNMENT

```
while(n > a){
```

$\lvert(n > a) \wedge (n \geq a) \wedge (a! = c)\rvert$	HYP and INV
$\lvert(n > a) \wedge (a! = c)\rvert$	IMPLIED
$\lvert(n \geq a + 1) \wedge ((a + 1)! = c * (a + 1))\rvert$	IMPLIED

```
a = a + 1;
```

$\lvert(n \geq a) \wedge (a! = c * a)\rvert$	ASSIGNMENT

```
b = x * b;
```

$\lvert(n \geq a) \wedge (a! = c * a)\rvert$	ASSIGNMENT

```
c = c * a;
```

$\lvert(n \geq a) \wedge (a! = c)\rvert$	ASSIGNMENT

```
}
```

$\lvert\neg(n > a) \wedge (n \geq a) \wedge (a! = c)\rvert$	PARTIAL WHILE
$\lvert(n = a) \wedge (a! = c)\rvert$	IMPLIED
$\lvert(n! = c)\rvert$	IMPLIED

```
}
```

$\lvert(n! = c)\rvert$	POST CONDITION

While in the first proof, $\lvert(n \geq a) \wedge (x^a = b)\rvert$ is chosen as the invariant to the while loop, $\lvert(n \geq a) \wedge (a! = c)\rvert$ is instead picked in the second proof. Both of these are legal invariants for this particular while loop. However, because the proof that is derived has a specific goal, the Hoare Logic approach forces the choosing of the right invariant for that goal, which is a difficult process to automate.

> "It is a necessary step [The discovery of an invariant] in order to use the while-rule and in general it requires intelligence and ingenuity. [...] Discovery of a suitable invariant requires careful thought about what the while statement is really doing "([17])

As illustrated in the example, a while loop can do (and often does) more than one thing. A Hoare Triple Logic proof, however does not conveniently support multiple invariance. While it is possible to construct multiple invariants using the $\wedge$ operator of the chosen logic (for example, we could use $|(n \geq a) \wedge (a! = c) \wedge (x^n = b)|$ as an invariant), this creates long expressions which are difficult to manage. The thread of the proof becomes harder to find, and in a formal setting, the use of the IMPLIED rule on one of the conjunctions would first require the use of some kind of $\wedge$ elimination step. However, if the invariant is multiple, then the prover might be unwilling to eliminate any of the conjunctions because it might contain information needed to complete the proof later.

There is another aspect of program correctness that could be improved on. In the example given above, there is no mechanism for reusing some of the work that was done in the first proof when the second proof is derived. Everything has to be redone from scratch. This increases the difficulty of implementing an efficient automatic system for correctness proving that would learn and reuse its acquired knowledge as it analyzes a particular code segment. One way to solve this problem is to subdivide programs into sub programs. Properties of these sub programs are then discovered and proved separately. and a correctness proof on a code segment results from putting together the different properties that have been derived about the sub programs. However, while the Hoare Triple proving techniques could easily be modified to use this subdivision approach, it is not formally integrated in the system. On the other hand, the system that we constructed is lemma based. This facilitates subdivision and reuse.

Finally, there is already a wide variety of implementations of first order logic theorem provers, but because of the Hoare Triple format which is very specific, it is difficult to use them to derive Hoare correctness proofs. The system that we built can easily be used in conjunction with existing tools. We demonstrate this in chapter 6.

1.2.3 Description of the Approach

Rather than work directly with a program and a defined computation, we translate a program into a logical theory and study the properties that follow from that theory.

For example, in the "func(x, n)" program from above, a map between input variables and output variables (the state of the local variables at the end of func()'s execution) can be represented by the following array :

(x, n)	(a, b, c)
$(0, 0)$	$(0, 1, 1)$
$(0, 1)$	$(1, 0, 1)$
$\vdots$	$\vdots$
$(1, 0)$	$(0, 1, 1)$
$(1, 1)$	$(1, 1, 1)$
$(1, 2)$	$(2, 1, 2)$
$\vdots$	$\vdots$
$(2, 0)$	$(0, 1, 1)$
$(2, 1)$	$(1, 2, 1)$
$(2, 2)$	$(2, 4, 2)$
$\vdots$	$\vdots$

This is a system, and it is ordered. Proving properties about that system, will then be equivalent to proving properties about the $func$ program. Proofs are then intended to show that certain properties of that system hold after the program has been executed. A logical system that allows this needs to be flexible enough to represent all the possible structures of an imperative programming language while supporting a classical deductive calculus (we constructed such a system, and we describe it in Chapter 4). Once a model has been created in such a logical system, we use the logic's proof calculus to derive proofs of its properties. This requires a precise description of the

relationship between the properties that are true of models and the properties that are true of the programs they are constructed from. We provide such a description in Chapter 5.
For the example above, one of the proving strategies that our approach could yield is as follows :

- The first step is to consider the body of the while loop as a program in its own right, with its own properties. Consider the code segment $funcA()$:

```
funcA()
{
        a = a + 1;
        b = x * b;
        c = c * a;
}
```

  1. It can be considered to be equivalent to the function $FA : \mathbb{N}^3 \Longrightarrow \mathbb{N}^3$ defined as $FA(u, v, w) = ((u + 1), (x * v), (w * (u + 1)))$, in that no matter what the state of the variable set $\{a, b, c\}$ at the beginning of $funcA()$'s execution is, it is provable that :

$$|(u = a) \wedge (v = b) \wedge (w = c)|\text{funcA}()|(a, b, c) = FA(u, v, w)|$$

     holds. Here, we use Hoare Triple Logic notation, and we will continue to do so throughout this thesis because it is convenient.

  2. Properties about the function FA can then be proven. The notion of *property* will be formally defined later. Once an equivalence relationship between the function FA and the programming construct *funcA()* is established, it can be shown how properties of the function FA translate into properties of the program *funcA()*; For example :

- Notice (and prove) that FA satisfies the following property :
  $\forall a, b, c, u, v, w \in \mathbf{N}$:
  $(((u, v, w) = FA(a, b, c)) \wedge (a! = c)) \longrightarrow (u! = w)$
- Notice (and prove) that FA satisfies the following propery :
  $\forall a, b, c, u, v, w \in \mathbf{N}$:
  $(((u, v, w) = FA(a, b, c)) \wedge (x^a = b)) \longrightarrow (x^u = v)$

Here, the symbol"$\longrightarrow$" is used as an abreviation of the meta language symbol for"implies". There is no formal definition attached to it (yet).

- The second step is to create *funcB()*:

```
funcB()
{
        a = 0;
        b = 1;
        c = 1;
}
```

1. *funcB* is equivalent to the constant function $FB : \mathbf{N}^3 \Longrightarrow \mathbf{N}^3$ defined as :
   $FB(u, v, w) = (0, 1, 1)$

2. The properties that we might want to prove about FB are :
   - $\forall a, b, c, u, v, w \in \mathbf{N}$: $((u, v, w) = FB(a, b, c)) \longrightarrow (u! = w)$
   - $\forall a, b, c, u, v, w \in \mathbf{N}$: $(((u, v, w) = FB(a, b, c)) \longrightarrow (x^u = v)$

- We would now look at the set of triples :

$$S = \{t : \mathbf{N}^3 | (x, y, z) = FA^m \circ FB(0, 0, 0), (0 \leq m)\}$$

where FA^m denotes the m-multiple composition of the FA function. We later prove a series of lemma (Chapter 5, Code-Equivalency of Composition, Code-Equivalency of SMP created expressions) that in

this case would allow us to conclude that no matter what the value of the variables x and n being passed to the function *void func(int x, int n)*, if when it finishes executing,

$$|\mathsf{T}|func|(a_e = a) \wedge (b_e = b) \wedge (c_e = c)|$$

holds (where given fixed x and n, (a_e, b_e, c_e) are the values of the variables a, b and c at the end of the programs execution) then :

$$\exists t_e \in S \text{ such that } t_e = (a_e, b_e, c_e)$$

For now, if we just assume this to be true, showing that :

1. For all(u, v, w) : $(u, v, w) \in S \longrightarrow (u! = w)$
2. For all(u, v, w) : $(u, v, w) \in S \longrightarrow (x^u = v)$

provides the desired result.

This is the general idea of what is formalized in this thesis.

Remark 1.2.1 In the second section of chapter 6, we go back to this example and prove a series of properties on the func program. We use a different more direct proof strategy (but the theory behind it is the same). We then link this strategy and the one presented in chapter 6 in our concluding remarks (chapter 7).

Remark 1.2.2 The above wouldn't be enough for a total correctness proof. In a total correctness, we would also need to show that the while loop ends eventually. This, as well as the separate issue of showing $a_e = n$ holds at the end of the program's execution are properties which are dependent on the conditional statement of the while loops, and its relationship with the body of the while loop. While in this thesis, we do not concern ourselves with those additional steps, concentrating instead on proving properties about relationships that are independent of the conditional statement, we discuss those steps in the future work section of chapter 7.

1.3 Contribution of the Thesis

The contribution of this thesis is :

- To build a logic designed for proof correctness.
- To present an algorithm which outputs a set of terms that characterize a program's computation in that logic using a program as input.
- To give a complete justification that the models constructed as above are sound.
- To provide an implementation of the algorithm showing its applicability.

1.4 The Thesis Organization

The thesis is organized as follows :

1. In chapter 1, the introduction to the thesis and its motivation was provided.
2. In chapter 2, the mathematical background needed for the work that was done is presented, and the set of properties that we propose our method can be used to prove a program computes is defined. We also show that the set we choose is as large as it can possibly be.
3. In chapter 3, a programming language is defined and it is proven that it can be used to compute every function corresponding to the properties defined in chapter 2.
4. In chapter 4, S_L logic, the logic that we built for the purpose of this thesis is formally presented.
5. In chapter 5, the algorithm that is used to represent programs in S_L logic is described. A justification of the soundness of the models that are constructed using this algorithm on the programs that were used as input is also given.

6. In chapter 6, an implementation of the approach is described, sample outputs and example proofs are presented.

7. In chapter 7, we compare with other work, conclude and discuss further work.

Chapter 2

Background and Function Set

In this chapter, the mathematical background of the thesis is presented. The terms *Tuple-Property* and *Tuple-Property Class* are defined and the class of properties upon which we would expect to be able to build proofs is introduced. A property is a mathematical function. We prove that a program or a program statement satisfies a property by showing that the values of the variables of the program or of the statement are always equal to the output of the property function (when the function's inputs are the values of the program's variables before the execution of the program).

2.1 Recursive Functions

In this section, we review some classical definitions of recursion theory.

Let $p \in \mathbf{N}$ be fixed. We call $\mathbb{F}_p$ the set of functions from $\mathbf{N}^p$ to $\mathbf{N}$

Definition 2.1.1 (Primitive Recursion[1]) Let $p \in \mathbf{N}$, $g \in \mathbb{F}_p$ and $h \in \mathbb{F}_{p+2}$ be fixed. A function $f \in \mathbb{F}_{p+1}$ is defined from g and h by *primitive recursion* if :

1. $f(x_1, x_2, ..., x_p, 0) = g(x_1, x_2, ..., x_p)$
2. $f(x_1, ..., x_p, y+1) = h(x_1, ..., x_p, y, f(x_1, ..., x_p, y))$

Definition 2.1.2 (μ-recursion[2]) Let $p \in \mathbf{N}$ be fixed. A function $f \in \mathbb{F}_p$ is defined from a relation R by *μ-recursion* if :

1. $\forall(x_1, ..., x_p) \in \mathbf{N}^p, \exists y \in \mathbf{N}$ such that $R(x_1, ..., x_p, y)$
2. $f(x_1, ..., x_p) = \mu y R(x_1, ..., x_p, y)$ where $\mu y R(x_1, ... x_p, y)$ is the least number y such that $R(x_1, ..., x_p, y)$ holds.

Similarly, $f \in \mathbb{F}_p$ is defined from $g \in \mathbb{F}_{p+1}$ by μ-recursion if :

1. $\forall(x_1, ..., x_p), \exists y \in \mathbf{N}$ such that $g(x_1, ..., x_p, y) = 0$
2. $f(x_1, \ldots, x_p) = \mu y(g(x_1, \ldots, x_p, y) = 0)$

Definition 2.1.3 (Recursive Functions[2]) The class of *recursive functions* is the smallest class of functions :

1. Containing the initial functions :

   (i) $\bigcirc(x) = 0$

   (ii) $S(x) = x + 1$

   (iii) $\partial_i^n(x_1, \ldots, x_n) = x_i$

2. Closed under composition,
   i.e., If $f_1, f_2, ... f_p, h \in \mathbb{F}_p$ are recursive functions then $f \in \mathbb{F}_p$, defined as $f(x_1, \ldots, x_p) = h(f_1(x_1, \ldots, x_p), f_2(x_1, \ldots, x_p), \cdots, f_p(x_1, \ldots, x_p))$ is also a recursive function.
3. Closed under primitive recursion and μ-recursion.

In this thesis we will only concern ourselves with natural numbers and functions operating on natural numbers. We will not consider cases of a function returning a negative number. This sometimes will be recalled, particularly when producing results related to μ-recursion.

Definition 2.1.4 (Characteristic functions) Let $A \subseteq \mathbf{N}^p$. The *characteristic function* $\chi_A : \mathbf{N}^p \Longrightarrow \{0, 1\}$ of A is defined as :

1. $\chi_A(a_1, a_2, \ldots, a_p) = 1$ if $(a_1, a_2, \ldots, a_p) \in A$

2. $\chi_A(a_1, a_2, \ldots, a_p) = 0$ otherwise

Definition 2.1.5 (Recursive sets) $A \subseteq \mathbb{N}^p$ is a *recursive set* if its characteristic function is recursive.

In ([11]), the interested reader will find a complete analysis of recursion theory on integers, as well as many annoted classical proofs and definitions. In order to give to the reader a clear idea of how we intend to use recursive functions, we now define some recursive functions that we will use often.

2.1.1 Some Recursive Functions

1. Addition is recursive. It can be defined by primitive recursion as :

$$\begin{cases} Add(x, 0) & = x \\ Add(x, y+1) & = S(Add(x, y)) \end{cases}$$

We will write the definition of Add a bit more formally than we will the other definitions:

(i) $S(x) = x + 1$ is recursive by definition.

(ii) $\partial_1^2(x_1, x_2) = x_1$ is recursive by definition.

(iii) $h(x_1, x_2, x_3) = S(\partial_3^3(x_1, x_2, x_3)) = x_3 + 1$ is recursive by composition.

(iv) Defining $Add(x, y)$ as :

$$\begin{cases} Add(x, 0) & = \partial_1^2(x, 0) \\ Add(x, y+1) & = h(x, y, Add(x, y)) \end{cases}$$

makes Add recursive by primitive recursion.

From now on, we write $x + y$ to mean $Add(x, y)$.

2. We define subtraction in two steps:

   (i) Defining $Pred(x)$ by primitive recursion as :

$$\begin{cases} Pred(0) & = 0 \\ Pred(x+1) & = x \end{cases}$$

   (ii) Allows us to define $\dot{-}$(x, y) as :

$$\begin{cases} \dot{-}(x,0) & = x \\ \dot{-}(x,y+1) & = Pred(\dot{-}(x,y)) \end{cases}$$

   And of course, we will write $x \dot{-} y$ for $\dot{-}(x, y)$.

3. We define multiplication :

$$\begin{cases} Mult(x,0) & = 0 \\ Mult(x,y+1) & = Add(x,(Mult(x,y))) \end{cases}$$

   We write $x * y$ for $Mult(x, y)$.

4. We define $IsntZero(x)$ as

$$IsntZero(x) := Sub(1, Sub(1, x))$$

5. We consider the set $Sm = \{(x, y) | x < y\}$ and we show that it is recursive by defining its characteristic function, $\chi_{Sm}(x, y)$ as

$$\chi_{Sm}(x, y) := IsntZero(Sub(y, x))$$

   We write $x < y$ to mean $\chi_{Sm}(x, y)$

6. We consider the set $Eq = \{(x, y) | x = y\}$ and we show that it is recursive by defining its characteristic function, $\chi_{Eq}(x, y)$ as

$$\chi_{Eq}(x, y) := Sub(1, IsntZero(Add(Sub(x, y), Sub(y, x))))$$

   We write $x = y$ to mean $\chi_{Eq}(x, y)$

7. We call a *projection function* $P_p^q : \mathbf{N}^q \Longrightarrow \mathbf{N}^p (p \leq q)$ any function which can be defined as :

   $$\{(\partial_{s_1}^p(x_1, \ldots, x_q), \ldots, \partial_{s_q}^p(x_1, \ldots, x_q)) | (x_1, \ldots, x_q) \in \mathbf{N}^q, \forall i, s_i \in \mathbf{N}, s_i < s_{i+1}, s_q \leq q\}$$

   Given $p, q \in \mathbf{N}$ such that $p \leq q$, a projection function $P_p^q : \mathbf{N}^q \Longrightarrow \mathbf{N}^p$, a recursive set $A \subseteq \mathbf{N}^p$, and 2 recursive functions f and $g \in \mathbb{F}_q$, we define the function $IfElse$ corresponding to

   $$\begin{cases} IfElse(x_1, \ldots, x_q) & = f(x_1, \ldots, x_q) \text{ if } P_p^q(x_1, \ldots, x_q) \in A \\ IfElse(x_1, \ldots, x_q) & = g(x_1, \ldots, x_q) \text{ otherwise} \end{cases}$$

   by composition as :

   $IfElse(x_1, \ldots, x_q) := Add(Mult(f(x_1, \ldots, x_q), \chi_A(P_p^q(x_1, \ldots, x_q))),$
   $Mult(g(x_1, \ldots, x_q), Sub(1, \chi_A(P_p^q(x_1, \ldots, x_q)))))$

It is also be proven that the predicates $\leq, >, \geq$, the functions *Exponentiation*, *Factorial* and *Fibonnaci*, and the boolean operators $\wedge, \longrightarrow, \vee, \neg$ are all examples of recursive constructs ([3] pg7-55).
Recursive functions give a nice delimitation to the notion of computability, which allows us to give a precise definition of the set of the functions on which it is suitable to apply our method.

2.2 Tuple-Properties

Definition 2.2.1 (Tuple-Property) Let $p \in \mathbf{N}$, A *Tuple-Property* is a pair of tuples : $< t = (t_1, t_2, \ldots, t_p), F = (f_1, \ldots, f_p) >$ satisfying :

$\forall i \leq p$, $f_i \in \mathbb{F}_p$ and is a recursive function and $t_i = f_i(t_1, t_2, \ldots, t_p))$.

Example 2.2.2 $< (3, 2, 6, 9), (f_1, f_2, f_3, f_4) >$ is a Tuple-Property where :

- $3 = f_1(3, 2, 6, 9)$, where $f_1(x_1, x_2, x_3, x_4) = \partial_1^4(x_1, x_2, x_3, x_4)$
- $2 = f_2(3, 2, 6, 9)$, where $f_2(x_1, x_2, x_3, x_4) = \partial_2^4(x_1, x_2, x_3, x_4)$

- $6 = f_3(3,2,6,9)$, where $f_3(x_1,x_2,x_3,x_4) = Fac(\partial_1^4(x_1,x_2,x_3,x_4))$
- $9 = f_4(3,2,6,9)$, where $f_4(x_1,x_2,x_3,x_4) = Exp(\partial_2^4(x_1,x_2,x_3,x_4), \partial_1^4(x_1,x_2,x_3,x_4))$

Definition 2.2.3 (Tuple-Property Class) Let $F = (f_1, f_2, \ldots, f_p) \in \mathbb{F}_p^p$ be a tuple of recursive functions. A *Tuple-property Class* on F is the set of Tuples-Properties

$$[T] = \{< t, F > | t \in \mathbb{N}^p, < t, F > \text{ is a Tuple-Property}\}$$

Tuple-Properties Classes are what we will show that a given program generates. The idea is to consider the state of the variables at the end of the program's execution as a tuple and to then prove that that tuple will always belong to a defined Class of Tuple-Properties. Properties are then proved about the Class.

Chapter 3

The C^c Programming Language

In this chapter, we will be defining formally the programming language upon which we wish to use the outlined correctness proving method. We will also describe some of its properties in relation to recursive functions and Property-Tuples.

3.1 Syntax

The language which we call C^c and which is a simplified version of the C programming language is defined as follows in Backus-Naur Form (BNF):

$$
\begin{aligned}
\boldsymbol{Num} &::= \boldsymbol{n}\ (n \in \mathrm{N}) \\
\boldsymbol{Cons} &::= \boldsymbol{c\_\{n\}}\ ,\ n \in \mathrm{N} \\
\boldsymbol{Var} &::= \boldsymbol{x\_\{n\}}\ ,\ n \in \mathrm{N} \\
\boldsymbol{Term} &::= \boldsymbol{Var|Cons|Term * Term|Term + Term|Term - Term|} \\
\boldsymbol{Expr1} &::= \boldsymbol{Var = Term;} \\
\boldsymbol{Expr2} &::= \boldsymbol{Term < Term|Term == Term} \\
\boldsymbol{IfStmt} &::= \boldsymbol{if(Expr2)\{ProgBody\}else\{ProgBody\}} \\
\boldsymbol{WhileStmt} &::= \boldsymbol{while(Expr2)\{ProgBody\}} \\
\boldsymbol{Stmt} &::= \boldsymbol{Expr1|IfStmt|WhileStmt} \\
\boldsymbol{ProgBody} &::= \boldsymbol{Stmt|Stmt[ProgBody]} \\
\boldsymbol{Prog} &::= \boldsymbol{ProgName(Cons[,Cons]*)\{ProgBody\}}
\end{aligned}
$$

Up to now, we haven't been very precise when talking about "code segments".

In the context of the of C^c language, a code segment is a *ProgBody* in the above BNF.

Because it is only meant to be a demonstration language for the method outlined in this work, C^c has been kept purposefully simple. We will show however that what we have is enough to compute every recursive function. which is all that we are aiming for.

Although the C and C^c languages have much in common. there are several differences between the two.

1. In C^c the names of the local variables and of the inputs in a given program are standardized. This facilitates the automatic derivation of mathematical functions from code segments.

2. C^c functions never return values. Because of this. any correctness proof on a C^c code segment must attempt to prove that the state of its local variables satisfy a given property at a given point in the program's execution.
   A proof then consists in showing the tuple produced by the variables declared inside a C^c program belongs to a given Tuple-Property class.

3. C^c is syntactically less powerful than C. It doesn't have any of the logical operators (&&, ||, !) found in C, misses a lot of the comparison operators ($>, \geq, \leq$) and doesn't have a case statement. It also lacks the"if" construction (C^c only has the "if/else" statement for branching). Finally, it doesn't support arrays, pointers and booleans (these can however be simulated easily by integers).

4. As far as evaluation rules, C^c doesn't recognize negative numbers and evaluates $x - y$ to 0 when $y \geq x$.

5. C^c assignment statements are labeled. The label we use are integers. the last assignment being labeled ([1]).

In theory however, because all of the above can be represented as recursive predicates or functions (except of course for pointers), C^c is computationally as powerful as the C language (by the lemma on the computability of recursive functions in C^c that will be presented later in the chapter). The proof system that we present in this thesis should still work if this extra "syntactic sugar" was to be added (future work).

Example 3.1.1 The following C^c program calculates the factorial of n. It is an output of the "NormalizeProgram" command of the Nivea system (our implementation of the method). "NormalizeProgram" takes a C program and translates it into C^c More details about Nivea will be given in Chapter 6.
This is the program that was given as input :

```
int Fac(int n)
{
        int  res  = 1;
        int  ctr = 0;

        while(ctr < n)
        {
                ctr = ctr + 1;
                res = res * ctr;
        }

        return res;
}
```

This is the Nivea C^c program output:

```
Vars :x_1 x_2 x_3
Return Var :x_3

Fac( n)
{
```

```
[5]x_1  = 1;
[4]x_2 = 0;
while(x_2 < n)
{
[3]x_2 = x_2 + 1;
[2]x_1 = x_1 * x_2;
}
[1]x_3 = x_1;
}
```

3.2 Hoare Triple Logic for C^c

In order to validate and compare the results we obtain using our method, we need to have at our disposal tools which allows us to provide correctness proofs independently of the method outlined in this thesis. We will use Hoare Triple Logic, which we will adapt to C^c. The rules of formation of the Hoare Triple Logic that we use can be found in [8] and in [17](pg 216-217). A similar logic is also presented in [18](pg 195-198).

Definition 3.2.1 (Hoare Triple Logic Partial Correctness) Let FUNC be a C^c code segment. We say that the triple $(|o|)$ FUNC $(|\psi|)$ is satisfied under *partial correctness* if, for all states which satisfy ϕ (o and ψ being some well formed expressions in a chosen logic), the state resulting from FUNC's execution satisfies the postcondition ψ, *provided that FUNC actually terminates.* In this case, we write:

$$\models_{\text{par}} (|\phi|) \text{ FUNC } (|\psi|)$$

This definition is taken verbatim from [17] (pg 226).

3.2.1 Hoare Triple Logic Rules for C^c

Given specifications for the C^c code segments FUNC1 and FUNC2, the Hoare Triple rules that we will be using were given at the beginning of chapter 1. References to these rules can be found in [8], [17] and [10]

3.3 Computing Recursive Functions in C^c

Since the general objective of this work is to develop a method which can be used to prove that a given recursive function is indeed computed by a given C^c program, we would like to know whether there are recursive functions that can not be computed by C^c programming. The next lemma shows that we need not worry about this.

Lemma 3.3.1 *(C^c **programs generate all recursive functions**) Let $p \in \mathbf{N}$ be fixed, $f \in \mathbb{F}_p$ a recursive function and $(n_1, n_2, ..n_p)$ a fixed tuple in $\mathbf{N}^p$. There exists a C^c program such that : $x_1 = f(n_1, ... n_p)$ at the end of the program's execution, where x_1 is the local variable that is declared first in the program.*

Proof. The proof is by recursion on the structure of f.

1. **Constant function**

   If $f = C(n) = c$ then we can use the program :

```
C(n)
{
    x_1 = [c];
}
```

   The following Hoare Triple proof structure:

|T|	PRE CONDITION

```
C(n)
{
```

|[c] = [c]|	IMPLIED

```
   x_1 = [c];
```

$\lvert x_1 = [c] \rvert$	ASSIGNMENT

}

$\lvert x_1 = [c] \rvert$	ASSIGNMENT

would provide a proof of the validity of the program.
In particular, this programming schema can be used to compute the $\bigcirc$ function where $\bigcirc(n) = 0$.

2. **Successor function**

   If $f = S(n) = n + 1$ then we can use the program :

```
s(n)
{
    x_1 = n;
    x_1 = x_1 + 1;
}
```

The following Hoare Triple proof :

|T|	PRE CONDITION

```
s(n)
{
```

$\lvert n = n \rvert$	IMPLIED
$\lvert n + 1 = n + 1 \rvert$	IMPLIED

```
   x_1 = n;
```

$\lvert x_1 + 1 = n + 1 \rvert$	ASSIGNMENT

```
   x_1 = x_1 + 1;
```

$|x_1 = n + 1|$ | ASSIGNMENT

}

$|x_1 = n + 1|$ | COPY

Provides the proof.

3. **Projection function**

If $f = \partial_i^p(n_1, n_2, \ldots, n_p) = n_i$ then we can use the program :

```
pi(n1, n2, ..., np)
{
    x_1 = [ni];
}
```

Where [ni] is replaced by the i^{th} parameter which is passed on to the function.
The following proof pattern :

|T| | PRE CONDITION

```
pi(n1, n2, ..., np)
{
```

$|n_i = n_i|$ | IMPLIED

```
   x_1 = [ni];
```

$|x_1 = n_i|$ | ASSIGNMENT

}

$x_1 = n_i$	COPY

will provide the result.

4. **Composition function**

Suppose f is defined by composition from $g_1, \ldots, g_k$ and h
i.e., $f(n_1, n_2, \ldots, n_p) = h(g_1(n_1, \ldots, n_p), g_2(n_1, \ldots, n_p), \ldots, g_k(n_1, \ldots, n_p))$.
and that by Induction Hypothesis, there are programs ASH, JAY1, JAY2, ..., JAYk satisfying the lemma for the functions $h, g_1, \ldots, g_k$.

Clearly, we can get each of the JAY1, ..., JAYk programs to run on the the input provided to the program that is going to compute f (which we'll call COMP). All we have to do is rename the occurences of the input parameters inside the JAY programs to get them to match those supplied to the COMP program. We will use this trick throughout the proof, but we won't state it anymore.

We can also reindex the local variables inside each of the JAY programs in order to leave some space for the variable x_1 in the COMP program and to assure that the variables used by these computations will not interfere with the other variables used by the program we intend to construct.

Finally, we can use the same trick used above to get the ASH program to run on the output supplied by each of the JAY programs and to create variable independence.

The resulting program looks like :

```
COMP(n1, n2, ..., np)
{
   x_1 = 0;

   [body of JAY1 running on input (n1, ... np) with          ]
   [local variables (x_1, x_2, ..., x_[G1])                  ]
   [reindexed to (x_2, x_3, ..., x_[G1 + 1])                 ]
```

```
   [body of JAY2 running on input (n1, ... np) with                    ]
   [local variables (x_1, x_2, ..., x_[G2])                            ]
   [reindexed to (x_[G1 + 2], x_[G1 + 3], ..., x_[G1 + G2 + 2])]
   .
   .
   .

   [body of JAYk running on input (n1, ... np) with                    ]
   [local variables (x_1, x_2, ..., x_[Gk]) reindexed to               ]
   [(x_[G1 + G2 + ... + G(k-1) + p],
                x_[G1 + G2 + ... + G(k-1) + p + 1], ...,               ]
   [            x_[G1 + G2 + ... + G(k-1) + Gk + p])                   ]


   [body of ASH running on input (x_2, x_[G1 + 2],
        x_[G1 + G2 + 3], ..., x_[G1 + G2 + ... + G(k-1) + p]) ]
   [with local variables (x_1, x_2, ..., x_[H]) reindexed to  ]
   [(x_[G1 + G2 + ... + G(k-1) + Gk + p + 1] , ...,
   [    x_[G1 + G2 + ... + G(k-1) + Gk + p + H])                       ]

   x_1 = x_[G1 + G2 + ... + G(k-1) + Gk + p + 1];
}
```

The following proof schema would then prove the correctness of the program.

$\|T\|$	PRE CONDITION

```
COMP(n1, n2, ..., np)

{
```

$\|0 = 0\|$	IMPLIED

```
x_1 = 0;
```

$$\frac{|x_1 = 0| \quad \text{ASSIGNMENT}}{|T|}$$

```
[body of JAY1 running on input $(n_1, \dots, n_p)$ with]
[JAY1 local variables $(x_1, x_2, \dots, x_{[[G1]]})     ]
[reindexed to $(x_2, x_3, ..., x_{[[G1 + 1]]})           ]
```

$|x_2 = g_1(n_1, \ldots, n_p)|$ (From Induction Hyp)

$$\frac{|(x_1 = 0) \wedge (x_2 = g_1(n_1, \ldots, n_p))| \quad \text{IMPLIED *}}{|T|}$$

```
[body of JAY2 running on input (n1, ... np) with          ]
[local variables (x_1, x_2, ..., x_[G2])                  ]
[reindexed to (x_[G1 + 2], x_[G1 + 3], ..., x_[G1 + G2 + 2])]
```

$|x_{[G1+2]} = g_2(n_1, \ldots, n_p)|$ (From Induction Hyp)

$$\frac{|(x_1 = 0) \wedge (x_2 = g_1(n_1, \ldots, n_p)) \wedge (x_{[G1+2]} = g_2(n_1, \ldots, n_p))| \quad \text{IMPLIED}}{}$$

.

.

.

$|T|$

```
[body of JAYk running on input (n1, ... np) with                    ]
[local variables (x_1, x_2, ..., x_[Gk]) reindexed to               ]
[(x_[G1 + G2 + ... + G(k-1) + p], x_[G1 + G2 + ... + G(k-1) + p + 1,]
[..., x_[G1 + G2 + ... + G(k-1) + Gk + p])                          ]
```

$|x_{[G1+G2+\ldots+G(k-1)+p]} = g_k(n_1, \ldots, n_p)|$ (From Induction Hyp)

$\|(x_1 = 0)$	
$\wedge(x_2 = g_1(n_1, \ldots, n_p))$	
$\wedge(x_{[G1+2]} = g_2(n_1, \ldots, n_p))$	
$\wedge \ldots$	
$\wedge(x_{[G1+G2+\ldots+G(k-1)+p]} = g_k(n_1, \ldots, n_p))\|$	IMPLIED
$\|T\|$	

```
[body of ASH running on input                                       ]
[(x_2, x_[G1 + 2], x_[G1 + G2 + 3], ...,
                                    x_[G1 + G2 + ... + G(k-1) + p])]
[with local variables (x_1, x_2, ..., x_[H]) reindexed to           ]
[(x_[G1 + G2 + ... + G(k-1) + Gk + p + 1],...,
                          x_[G1 + G2 + ... + G(k-1) + Gk + p + H])]
```

$|x_{[G1+G2+\ldots+G(k-1)+Gk+p+1]} = h((x_2, x_{[G1+2]}, x_{[G1+G2+3]}, \ldots, x_{[G1+G2+\ldots+G(k-1)+p]}))|$

$\|(x_1 = 0)$	
$\wedge(x_2 = g_1(n_1, \ldots, n_p))$	
$\wedge(x_{[G1+2]} = g_2(n_1, \ldots, n_p))$	
$\wedge \ldots$	
$\wedge(x_{[G1+G2+\ldots+G(k-1)+p]} = g_k(n_1, \ldots, n_p))$	
$\wedge(x_{[G1+G2+\ldots+G(k-1)+Gk+p+1]} =$	
$\quad h((x_2, x_{[G1+2]}, x_{[G1+G2+3]}, \ldots, x_{[G1+G2+\ldots+G(k-1)+p]}))\|$	IMPLIED
$\|x_{[G1+G2+\ldots+G(k-1)+Gk+p+1]} =$	
$\quad h(g_1(n_1, \ldots, n_p), g_2(n_1, \ldots, n_p), \ldots, g_k(n_1, \ldots, n_p))\|$	IMPLIED

```
x_1 = x_[G1 + G2 + ... + G(k-1) + Gk + p + 1];
```

$\lvert x_1 = h(g_1(n_1,\ldots,n_p), g_2(n_1,\ldots,n_p),\ldots,g_k(n_1,\ldots,n_p))\rvert$	ASSIGNMENT

```
}
```

$\lvert x_1 = h(g_1(n_1,\ldots,n_p), g_2(n_1,\ldots,n_p),\ldots.g_k(n_1,\ldots.n_p))\rvert$	COPY

5. **Primitive recursion**

   If f is defined by primitive recursion, then

   (i) $f(n_1, n_2, \ldots, n_p, 0) = g(n_1, n_2, \ldots, n_p)$

   (ii) $f(n_1, \ldots, n_p, y+1) = h(n_1, \ldots, n_p, y, f(n_1, \ldots. n_p, y))$

   Where by induction hypothesis, there are C^c programs which compute g and h (respectively JAY and ASH) then we consider the following programming structure :

```
REC(n1, n2, ..., np, y)
{
    x_1 = 0;
    x_2 = 0;

    [Body of JAY running on input (n1, n2, ..., np) with JAY local]
    [variables (x_1, ..., x_[G] reindexed to (x_3, ...x_[G + 2])  ]
    [in REC                                                       ]

    x_1 = x_3;

    while(x_2 < y)
    {
        x_2 = x_2 + 1;

        [Body of ASH running on input (n1, n2, ..., np, x_2-1, x_1) ]
        [with ASH local variables (x_1, ..., x_[H]) reindexed to    ]
```

```
        [x_[G + 3], ... x_[G + 3 + H]) in REC                    ]

        x_1 = x_[G + 3];
    }
}
```

Remark 3.3.2 Note that the input variables to ASH are the renamed x_1 from each of the JAY programs.

Which calculates f as shown by the following proof strategy:

$\|T\|$	PRE CONDITION

```
REC(n1, n2, ..., np, y)

{
```

$\|f(n_1, n_2, \ldots, n_p, 0) = g(n_1, n_2, \ldots, n_p)\|$	IMPLIED

```
    x_1 = 0;
```

$\|f(n_1, n_2, \ldots, n_p, 0) = g(n_1, n_2, \ldots, n_p)\|$	IMPLIED

```
    x_2 = 0;
```

$\|f(n_1, n_2, \ldots, n_p, x_2) = g(n_1, n_2, \ldots, n_p)\|$	ASSIGNMENT

$|T|$

```
   [Body of JAY running on input (n1, n2, ..., np)       ]
   [with local variables(x_1, ..., x_[G] reindexed       ]
   [to (x_3, ...x_[G + 2])                               ]
```

$|x_3 = g(n_1, n_2, \ldots, n_p)|$ (By Induction Hyp)

$\|(f(n_1, n_2, \ldots, n_p, x_2) = g(n_1, n_2, \ldots, n_p)) \wedge (x_3 = g(n_1, n_2, \ldots, n_p))\|$	IMPLIED
$\|(f(n_1, n_2, \ldots, n_p, x_2) = x_3)\|$	IMPLIED

```
x_1 = x_3;
```

$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = x_1)\rvert$	ASSIGNMENT

```
while(x_2 < y)

{
```

$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \wedge (x_2 < y)\rvert$	INVARIANT HYP and GUARD
$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = x_1)$ $\wedge(f(n_1, n_2, \ldots, n_p, x_2 + 1) =$ $h(n_1, n_2, \ldots, n_p, x_2, f(n_1, n_2, \ldots, n_p, x_2))\rvert$	IMPLIED
$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = x_1)$ $\wedge(f(n_1, n_2, \ldots, n_p, x_2 + 1) = h(n_1, n_2, \ldots, n_p, x_2, x_1)\rvert$	IMPLIED
$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = x_1)$ $\wedge(f(n_1, n_2, \ldots, n_p, x_2 + 1) =$ $h(n_1, n_2, \ldots, n_p, x_2 + 1 - 1, x_1)\rvert$	IMPLIED

```
    x_2 = x_2 + 1;
```

$\lvert f(n_1, n_2, \ldots, n_p, x_2) = h(n_1, n_2, \ldots, n_p, x_2 - 1, x_1)\rvert$	ASSIGNMENT

$\lvert T \rvert$

```
[Body of ASH running on input (n1, n2, ..., np, x_2-1, x_1)    ]
[with local ASH variables (x_1, ..., x_[H]) reindexed to       ]
[x_[G + 3], ..., x_[G + 3 + H])                                ]
```

$\lvert x_{[G+3]} = h(n_1, n_2, ..., n_p, x_2 - 1, x_1)\rvert$ (From Induction Hyp)

$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = h(n_1, n_2, \ldots, n_p, x_2 - 1, x_1))$ $\wedge(x_{[G+3]} = h(n_1, n_2, \ldots, n_p, x_2 - 1, x_1))\rvert$	IMPLIED
$\lvert(f(n_1, n_2, \ldots, n_p, x_2) = x_{[G+3]})\rvert$	IMPLIED

```
    x_1 = x_[G + 3];
```

$\lvert f(n_1, n_2, \ldots, n_p, x_2) = x_1 \rvert$	ASSIGNMENT

```
}
```

$\lvert (f(n_1, n_2, \ldots, n_p, x_2) = x_1)) \wedge (x_2 \leq y) \wedge \neg(x_2 < y) \rvert$	PARTIAL-WHILE
$\lvert (f(n_1, n_2, \ldots, n_p, x_2) = x_1)) \wedge (x_2 = y) \rvert$	IMPLIED
$\lvert f(n_1, n_2, \ldots, n_p, y) = x_1 \rvert$	IMPLIED
$\lvert x_1 = f(n_1, n_2, \ldots, n_p, y) \rvert$	IMPLIED

```
}
```

$\lvert x_1 = f(n_1, n_2, \ldots, n_p, y) \rvert$	COPY

6. **Mu-recursion**

If $f(n_1, n_2, \ldots, n_p) = \mu y(g(n_1, n_2, \ldots, n_p, y) = 0)$
We use the following program and associated proof structure.

```
MREC(n1, n2, ..., np)
{
    x_1 = 0;

    x_2 = 0;

    [Body of JAY running on input          ]
    [(n1, n2, ..., np, x_1) with local     ]
    [variables (x_1, ..., x_[G]) reindexed ]
    [to (x_3, ...x_[G + 2])]               ]

    x_2 = x_3;


    while(0 < x_2)
```

```
    {
        x_1 = x_1 + 1;

        [Body of JAY running on input          ]
        [(n1, n2, ..., np, x_1) with local     ]
        [variables (x_1, ..., x_[G]) reindexed ]
        [to (x_[G + 3], ...,x_[G + G + 3])     ]

        x_2 = x_[G + 3];
     }

}
```

Which we look in the context of the following proof format :

$\|T\|$	PRE CONDITION

```
MREC(n1, n2, ..., np)
{
```

$\|\neg\exists x \in \mathbf{N}(0 \leq x < 0)\|$	IMPLIED
$\|\forall x \in \mathbf{N}(0 \leq x < 0) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)\|$	IMPLIED

```
    x_1 = 0;
```

$\|\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)\|$	ASSIGNMENT

```
    x_2 = 0;
```

$\|\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)\|$	COPY

$|T|$

```
    [Body of JAY running on input          ]
    [(n1, n2, ..., np, x_1) with local     ]
    [variables (x_1, ..., x_[G]) reindexed ]
    [to (x_3, ...x_[G + 2])]               ]
```

$|x_3 = g(n_1, n_2, ..., n_p, x_1)|$ (From Induction Hyp)

$\|(x_3 = g(n_1, n_2, ..., n_p, x_1))$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0))\|$	IMPLIED

```
x_2 = x_3;
```

$\|(x_2 = g(n_1, n_2, ..., n_p, x_1))$ $\wedge\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0))\|$	ASSIGNMENT

```
while(0 < x_2)
{
```

$\|(x_2 = g(n_1, n_2, ..., n_p, x_1))$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0))$ $\wedge(0 < x_2)\|$	INVARIANT and GUARD
$\|(x_2 = g(n_1, n_2, ..., n_p, x_1))$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0))$ $\wedge(x_2 <> 0)\|$	IMPLIED
$\|g(n_1, n_2, ..., n_p, x_1) <> 0)$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0))\|$	IMPLIED
$\|\forall x \in \mathbf{N}(0 \leq x \leq x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)\|$	IMPLIED
$\|\forall x \in \mathbf{N}(0 \leq x < x_1 + 1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)\|$	IMPLIED

```
x_1 = x_1 + 1;
```

$\|\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)\|$	ASSIGNMENT

$|T|$

```
[Body of JAY running on input          ]
[(n1, n2, ..., np, x_1) with local     ]
[variables (x_1, ..., x_[G]) reindexed ]
[to (x_[G + 3], ...,x_[G + G + 3])     ]
```

$|(x_{[G+3]} = g(n_1, n_2, ..., n_p, x_1) \wedge (x_{[G+3]} \geq 0)|$ From Induction Hyp

$\|(x_{[G+3]} = g(n_1, n_2, ..., n_p, x_1) \wedge (x_{[G+3]} \geq 0)$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p. x) <> 0))\|$	IMPLIED

```
x_2 = x_[G + 3];
```

$\|(x_2 = g(n_1, n_2, ..., n_p, x_1) \wedge (x_2 \geq 0)$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p. x) <> 0))\|$	ASSIGNMENT

}

$\|(x_2 = g(n_1, n_2, ..., n_p, x_1)) \wedge (x_2 \geq 0)$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)))$ $\wedge\neg(0 < x_2)\|$	PARTIAL-WHILE
$\|(x_2 = g(n_1, n_2, ..., n_p, x_1)) \wedge (x_2 = 0)$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p. x) <> 0)))\|$	IMPLIED
$\|(g(n_1, n_2, ..., n_p, x_1) = 0)$ $\wedge(\forall x \in \mathbf{N}(0 \leq x < x_1) \longrightarrow (g(n_1, \ldots, n_p, x) <> 0)))\|$	IMPLIED
$\|x_1 = \mu y(g(n_1, n_2, ..., n_p, y) = 0)\|$	IMPLIED
$\|x_1 = f(n_1, n_2, ..., n_p)\|$	IMPLIED

}

$|x_1 = f(n_1, n_2, ..., n_p)|$ | COPY

□

Corollary 3.3.3 (C^c Programs generate all Tuple-properties) *(plus some extra variables)*

Proof.
The proof is a simple application of the above lemma : Let $T = (t_1, t_2, \ldots, t_p)$ be a tuple property with associated function set $F = (f_1, \ldots, f_p)$. Because all the functions forming F are recursive by definition, by the lemma above (3.3.1), we know that for all i, $1 \leq i \leq p$ there is a C^c program (TEEi) such that on input $(t_1, t_2, \ldots, t_p)$ the local variable x_1 of TEEi is equal to $f_i(T)$ at the end of the program's execution.
We then construct the following C^c program using the TEE programs :

```
TUP(t_1, t_2, ..., t_p)
{
        x_1 = 0;
        x_2 = 0;
        x_3 = 0;
        .
        .
        .
        x_[p] = 0;

        [Body of TEE1 program running on input (t_1, t_2, ..., t_p)
        with local variables (x_1, x_2, ..., x_[T1]) reindexed to
        (x_[p + 1], x_[p + 2], ..., x_[p + T1])]

        [Body of TEE2 program running on input (t_1, t_2, ..., t_p)
        with local variables (x_1, x_2, ..., x_[T2]) reindexed to
        (x_[p + T1 + 1], x_[p + T1 + 2], ..., x_[p + T1 + T2])]
        .
        .
        .
```

```
        [Body of TEEp program running on input (t_1, t_2, ..., t_p)
        with local variables (x_1, x_2, ..., x_[T1]) reindexed to
        (x_[p + T1 + ... + T(p-1) + 1], x_[p + T1 + ... + T(p-1) + 2], ...,
        x_[p + T1 + ... + T(p-1) + Tp])]


        x_1 = x_[p + 1];
        x_2 = x_[p + T1 + 1];
        .
        .
        .
        x_p = x_[p + T1 + ... + T(p-1) + 1];

}
```

The proof is then complete by the fact that $(x_1, x_2, \ldots, x_p) = (t_1, t_2, \ldots, t_p)$.
□

Chapter 4

The S_L Logic

Simple Logic (S_L) is the formal system that was built for the purpose of this thesis. It is a first order logic with arithmetic whose main intended use is to reason about recursive functions built from primitives. Induction is built in to the rules of inference.

4.1 S_L, the Formal Language Overview

A formal system is typically built from three components :

- A vocabulary, complete with rules of formation, that allows the identification of the well-formed formulas of the system.
- A set of rules that can be used on the well-formed formulas of the language in order to deduce new well-formed formulas.
- A set of well-formed formulas called *axioms* that need not be proven when used in the derivation of new formulas.

In the designing of S_L, there were some informal constraints that were taken into account :

1. Because of the use for which the system was intended, it needed to be easy to model multiple variable computations using S_L.

2. Because induction is the main proving strategy that is used to prove properties of code segments, S_L needed to support induction.

The S_L language is built on :

1. An infinite set of function symbols $\Gamma = \{A, B, ... Z, AA, AB, ...\}$
2. An infinite set of variable symbols $\chi = \{x_1, x_2, \ldots\}$
3. The set of numerals associated with the natural numbers : $\{0, 1, 2, \ldots\}$
4. The set of symbols $\{ind, +, -, *, (,), <, =, \vee, \wedge, \longrightarrow, \forall\}$

The symbol *ind* is a constant whose main use is in the induction derivations.

4.2 Syntax

A formal description of the rules of formation in their Backus-Naur Form (BNF) is now presented. The usual symbols representing regular languages operators are modified because the symbols sometimes clash with the symbols that are used in S_L.

4.2.1 Regular Language Operators:

We will define the logic using the following regular language operators.

- $[X]!$: Matches 0 or 1 instance of X.
- $[X]+'$: Matches 1 or more instances of X.
- $[X]*'$: Matches 0 or more instances of X.
- $[Y|X]$: Matches X or Y. The brackets are omited when unnecessary.

4.2.2 Terms

The possible terms and sentences of the logic are labeled as being of type A, B, C, D, E, F.

Terms of type A, B, C and D

$$A ::= x|ind|m|(A * A)|(A + A)|(A - A) \text{ where } x \in \chi, m \in \mathbf{N}$$

Examples :

'4', '$(x_1 - x_2)$', '$((y_1 + ind) - (x_2 + 2))$' are examples of type A terms.

$$B ::= (A[,A]*')$$

Example :

'$((y_1 * 3), 4, ((x_1 + x_2) * 2))$' is a type B term

$$C ::= A[,C] *' \mid B[,C] *' \mid XB[,C] *' \text{ where } X \in \Gamma$$

Examples :

'$(y_1 * (5 + x_1)), ((x_1 + x_2) * 2)$'
'$FAC((n + 1), 4)$'
are type C terms.

$$D ::= B \mid X(C) \mid X(D[,D]*') \text{ where } X \in \Gamma$$

Examples :

Below are some type D terms with their derivation trees.

- '(y_1, y_2, y_3)'

$$(y_1, y_2, y_3)$$

$$(A, A, A)$$
$$B$$
$$D$$

- '$FUNC(EXP((n+1), 4), 4, (y_1 + 2))$'

$$FUNC(EXP((n+1). 4), 4, (y_1 + 2))$$
$$FUNC(EXP(A. A). A. A)$$
$$FUNC(EXPB, A. A)$$
$$FUNC(C)$$
$$D$$

- '$FUNC(APPL((n+1). 4). 4, (x1, x_2, x_1), POL1(x_2). FUNC2((y_1+2)))$'

$$FUNC(APPL((n+1), 4), 4. (x_1, x_2, x_1). POL1(x_2). FUNC2((y_1+2)))$$
$$FUNC(APPL(A. A). A, (A, A. A). POL1(A). FUNC2(A))$$
$$FUNC(APPLB, A, B, POL1B. FUNC2B)$$
$$FUNC(C)$$
$$D$$

- '$FUNC1(FUNC2(FUNC3(x_1, x_2, x_3). n))$'

$$FUNC1(FUNC2(FUNC3(x_1, x_2, x_3). n))$$
$$FUNC1(FUNC2(FUNC3(A, A, A), A))$$
$$FUNC1(FUNC2(FUNC3B, A))$$
$$FUNC1(FUNC2(C))$$
$$FUNC1(D)$$
$$D$$

Sentences of type E and F

$$E ::= < D = D > \mid < A = A > \mid < A < A >$$

Examples :

'$< (x_1, x_2, x_3) = FUNC1(FUNC2(0, y_1, 3)) >$'
'$< FUNC1(x_1) = FUNC2(x_2, x_3) >$'
'$< (x_1, x_2, x_3) = (x_1, x_2, x_3, x_4) >$'
are examples of type E sentences. Note that while the last is also considered to be a valid type E sentence, it would probably be false in most well-formed S_L models.

$$F ::= E|(\neg F)|(F \vee F)|(F \wedge F)|(F \longrightarrow F)|\forall(x) : F|\exists(x) : F \text{ where } x \in \backslash$$

Informally, type F sentences are sometimes referred to as 'expressions'.

Notation :

- Any valid expression matching $(F[\wedge F] +' \mid [F[\vee F]+')$ can be parenthesized in any manner consistent with binding conventions.
- $(\forall(x_1) : (\forall(x_2) : (\ldots \forall(x_n) : F)) \ldots)$ is abbreviated $\forall(x_1, x_2, \ldots, x_n) : F$
- $(\exists(x_1) : (\exists(x_2) : (\ldots \exists(x_n) : F)) \ldots)$ is abbreviated $\exists(x_1, x_2, \ldots, x_n) : F$

4.3 S_L, The Rules of Inference

In [7], the notion of proof is defined thus :

> "A **(formal) proof** is a finite column $< S_1, S_2, \ldots, S_k >$ of statements of the theory such that each S either is an axiom or comes from one or more preceding S's by the rules of inference of the system of logic employed. A theorem or provable statement is a statement which is the last line of some proof."

In order to be able to derive statements from a set of S_L statements. there needs to be rules of inference to apply to the statements. These are given below.

Notation

Once again, the rules are described using a regular expression machine. It uses the same rules as the ones introduced above with some additions :

1. It matches the variables A, B, C, D, E, F respectively with terms or sentences of type A, B, C, D, E, F : x, y, z with elements of γ: X, Y, Z with elements of Γ and n, m with elements of **N**.

2. The matching variables use subscripts to match the i_{th} expression of a term or sentence : A_4 matches the 4^{th} sub expression of type A.

3. $S\{h => c\}$ is taken to mean replace all occurrences of h by c in the expression S.

4. Regular expression matching variables #1. #2. #3.... are used to represent any match.

5. Finally, it is always assumed that the terms or sentences above and below the rule line are in the valid S_L subset of all the regular expressions that they could match.

A tableau style of proof is used, where each line in the proof is numbered vertically, and the proof is derived from the expressions that are written above. Given a set $\Sigma = \{\phi_1, \phi_2, \cdots, \phi_n\}$, we say that γ is provable from Σ if one of the following applies:

- $\gamma \in \Sigma$
- There is a proof which derives γ using only $\phi_1, \phi_2, \cdots, \phi_n$ as its premisses

In that case, we write :

$$\Sigma \vdash \gamma$$

4.3.1 S_L Inference Rules

- $\wedge$ rules :
  The $\wedge$ rules we present here are derivable from the usual $\wedge$ rules of predicate logic. We prefer to use this version to simplify proof presentations.

  1.
  $$\frac{F_1 \quad F_2 \quad \ldots \quad F_p}{(F_1 \wedge F_2 \wedge \ldots F_p)} (\wedge \text{ Intro })$$

  2.
  $$\frac{F_1 \wedge F_2 \wedge \quad \ldots \quad \wedge F_p}{F_i} (\wedge \text{ Elim })$$

  Where $(1 \leq i \leq p)$.

- $\vee$ rules :
  The $\vee$ rules we present here are derivable from the usual $\vee$ rules of predicate logic. Again, we use this version to simplify proof presentations.

  1.
  $$\frac{F_i}{(F_1 \vee \ldots F_i \ldots \vee F_p)} (\vee \text{ Intro })$$

  2.
  $$\frac{\begin{array}{cccccc} & & (F_1) & (F_2) & & (F_p) \\ & & \vdots & \vdots & & \vdots \\ F1 \vee F2 \ldots \vee F_p & & F & F & \ldots & F \end{array}}{F} (\vee \text{ Elim })$$

  Where $F_1 \ldots F_p$ are *Assumptions* that are discharged by this rule's application.

- $\longrightarrow$ rules :

1.

$$\frac{\begin{array}{c}(F_1)\\ \vdots \\ F_2\end{array}}{F_1 \longrightarrow F_2}\ (\longrightarrow \text{ Intro })$$

Where F_1 is an *Assumption* that is discharged by this rule's application.

2.

$$\frac{F_1 \longrightarrow F_2 \qquad F_1}{F_2}\ (\longrightarrow \text{ Elim })$$

- $=$ Rules :

1.

$$\frac{}{< A_1 = A_1 > \mid < D_1 = D_1 >}\ (= \text{ Intro })$$

2.

$$\frac{< (\#1, \#2, \ldots, \#p) = (\#1', \#2', \ldots, \#p') >}{< \#i = \#i' >}\ (= \text{ Elim })$$

Where $1 \leq i \leq p$.

3.

$$\frac{< \#1 = \#2 > \qquad F\{x => \#1\}}{F\{x => \#2\}}\ (= Sub)$$

- $\forall$ rules :

1.

$$\frac{\begin{array}{c}(x)\\ \vdots \\ F\end{array}}{\forall(x) : F}\ (\forall \text{ Intro })$$

Where x is a variable that doesn't occur freely previously in the proof.

Notation :

Multiple applications of the ∀ Intro rule are written :

$$\frac{\begin{array}{c}(x_1, x_2, \ldots, x_p) \\ \vdots \\ F\end{array}}{\forall(x_1, x_2, \ldots, x_p) : F} (\forall \text{ Intro })$$

2.

$$\frac{\forall(x) : F}{F\{x => A\}} (\forall \text{ Elim })$$

Note that it is now possible to derive:

$$\frac{\forall(x_1, x_2, \ldots, x_p) : F}{\forall(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p) : F\{x_i => A\}} (\forall \text{ Elim })$$

Provided that all $x_1, \ldots, x_p$ are distinct variables and that there are no free occurrences of $(x_1, \ldots, x_{i-1}, x_{i+1}, x_p)$ in A.

- ∃ rules :

1.

$$\frac{\forall(x) : F \qquad \begin{array}{c}(x_0)F\{x => x_0\} \\ \vdots \\ F_1\end{array}}{F_1} (\exists \text{ Elim })$$

where x_0 does not occur in F_1.

- Rename rule :

$$\frac{\forall(x_1, x_2, \ldots, x_p) : F}{\forall(y_1, y_2, \ldots, y_p) : F\{x_1 => y_1, x_2 => y_2, \ldots, x_p => y_p\}} (\text{ RENAME })$$

Provided that all $x_1, \ldots, x_p$ and $y_1, \ldots, y_p$ are distinct variables.

This rule is only applicable when there are no y_i's occurring in F. It follows the usual rules of bound/free variable substitutions in formal languages with $\forall$ being the binding operator.

Enough rules have now been provided to derive some example proofs.

Examples :

Let the set of premisses Σ be defined as:

Tag	Premiss
A5	$\forall(x_1,x_2,x_3): < AFAC5(x_1,x_2,x_3,n) = (1,x_2,x_3,n) >$
A4	$\forall(x_1,x_2,x_3): < AFAC4(x_1,x_2,x_3,n) = (x_1,0,x_3,n) >$
A3	$\forall(x_1,x_2,x_3): < AFAC3(x_1,x_2,x_3,n) = (x_1,x_2+1,x_3,n) >$
A2	$\forall(x_1,x_2,x_3): < AFAC2(x_1,x_2,x_3,n) = (x_1 * x_2,x_2,x_3,n) >$
A1	$\forall(x_1,x_2,x_3): < AFAC1(x_1,x_2,x_3,n) = (x_1,x_2,x_1,n) >$
G2	$\forall(x_1,x_2,x_3): < GFAC2(x_1,x_2,x_3,n) = AFAC4(AFAC5(x_1,x_2,x_3,n)) >$
G1	$\forall(x_1,x_2,x_3): < GFAC1(x_1,x_2,x_3,n) = AFAC2(AFAC3(x_1,x_2,x_3,n)) >$
W1a1	$\forall(x_1,x_2,x_3,x_4): < (x_4 = 0) >$
	$\longrightarrow < WFAC1A(x_1,x_2,x_3,n,x_4) = (x_1,x_2,x_3,n) >$
W1a2	$\forall(x_1,x_2,x_3,x_4): < (0 < x_4) >$
	$\longrightarrow < WFAC1A(x_1,x_2,x_3,n,x_4) = GFAC1(WFAC1A(x_1,x_2,x_3,n,x_4-1)) >$
W1a3	$\forall(x_1,x_2,x_3,x_4): \exists(v_1,v_2,v_3):$
	$< (v_1,v_2,v_3,n) = WFAC1A(x_1,x_2,x_3,n,x_4) >$

Proof 1 :

The first example proof (named [D1]) shows how a simple property of a composed function can be proven. [D1], as well as the following proofs of this section were made and checked without the use of any tools. They are just meant to be examples. We chose these particular examples because they are similar in structure to the kind of proofs that would have to be derived in a program correctness setting.

Proving [D1] :

$\forall(y_1,y_2,y_3,x_1,x_2,x_3) :< (y_1,y_2,y_3,n) = GFAC2(x_1,x_2,x_3,n) >$

$\longrightarrow < (y_1,y_2,y_3,n) = (1,0,x_3,n) >$

Proof 1:

1.[Variables] $(y_1, y_2, y_3, x_1, x_2, x_3)$
2. [Assumption] $< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >$
3. [G2 RENAME] $\forall(z_1, z_2, z_3) :< GFAC2(z_1, z_2, z_3, n) > = AFAC4(AFAC5(z_1, z_2, z_3, n)) >$
4. [$\forall$ Elim, 3] $< GFAC2(x_1, x_2, x_3, n) = AFAC4(AFAC5(x_1, x_2, x_3, n)) >$
5. [A5 RENAME] $\forall(z_1, z_2, z_3) :< AFAC5(z_1, z_2, z_3, n) = (1, z_2, z_3, n) >$
6. [$\forall$ Elim, 5] $< AFAC5(x_1, x_2, x_3, n) = (1, x_2, x_3, n) >$
7. [= Sub 4, 6] $< GFAC2(x_1, x_2, x_3, n) = AFAC4(1, x_2, x_3, n) >$
8. [A4 RENAME] $\forall(z_1, z_2, z_3) :< AFAC4(z_1, z_2, z_3, n) = (z_1, 0, z_3, n) >$
9. [$\forall$ Elim 8] $< AFAC4(1, x_2, x_3, n) = (1, 0, x_3, n) >$
10. [= Sub 7, 9] $GFAC2(x_1, x_2, x_3, n) = (1, 0, x_3, n) >$
11. [= Sub 10, 2] $< (y_1, y_2, y_3, n) = (1, 0, x_3, n) >$
12. [$\longrightarrow$ Intro 2,3-11] $< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) > \longrightarrow < (y_1, y_2, y_3, n) = (1, 0, x_3, n) >$
13.[$\forall$ Intro 1.2-12] $\forall(y_1, y_2, y_3, x_1, x_2, x_3) :< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) > \longrightarrow < (y_1, y_2, y_3, n) = (1, 0, x_3, n) >$

Proof 2 :

The next proof shows how we can use [D1] to prove that the $GFAC2$ function satisfies a more complex property. Namely, that any output (y_1, y_2, y_3, n) of $GFAC2$ will have $y_1 = fac(y_2)$.

Proving [D2] :

$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >$

$\longrightarrow < y_1 = fac(y_2) >$

Where fac is the factorial function.

Proof 2:

1.[Variables]$(y_1, y_2, y_3, x_1, x_2, x_3)$
2. [Assumption]$< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >$
3. [D1 RENAME]$\forall(z_1, z_2, z_3, w_1, w_2, w_3) :< (z_1, z_2, z_3, n) = GFAC2(w_1, w_2, w_3, n) >$
$\longrightarrow< (z_1, z_2, z_3, n) = (1, 0, w_3, n) >$
4. [$\forall$ Elim 3]$< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >$
$\longrightarrow< (y_1, y_2, y_3, n) = (1, 0, x_3, n) >$
5. [$\longrightarrow$ Elim 2, 4]$< (y_1, y_2, y_3, n) = (1, 0, x_3, n) >$
6. [Number Theory]$< 1 = fac(0) >$
7. [= Elim 5]$< y_2 = 0 >$
8. [= Sub 6, 7]$< 1 = fac(y_2) >$
9. [= Elim 5]$< y_1 = 1 >$
10. [= Sub 9, 8]$< y_1 = fac(y_2) >$
11. [$\longrightarrow$ Intro 2, 3-10]$< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >\longrightarrow< y_1 = fac(y_2) >$
12.[$\forall$ Intro 1, 2-11]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >$
$\longrightarrow< y_1 = fac(y_2) >$

The line 6 deduction is justified with the label : [Number Theory]. the meaning will be clarified later in this chapter.

Proof 3 :

We now prove that $GFAC1$ preserves the same factorial property from above in its output.

Proving [D3] :

$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$

$(< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > \wedge < x_1 = fac(x_2) >)$

$\longrightarrow< y_1 = fac(y_2) >$

Proof 3:

1.[Variables]$(y_1, y_2, y_3, x_1, x_2, x_3)$

2. [Assumption]$(< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > \wedge < x_1 = fac(x_2) >)$

3. [$\wedge$ Elim 2]$< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) >$

4. [$\wedge$ Elim 2]$< x_1 = fac(x_2) >$

5. [G1 RENAME]$\forall(z_1, z_2, z_3) :< GFAC1(z_1, z_2, z_3, n) = AFAC2(AFAC3(z_1, z_2, z_3, n)) >$

6. [$\forall$ Elim 5]$< GFAC1(x_1, x_2, x_3, n) = AFAC2(AFAC3(x_1, x_2, x_3, n)) >$

7. [A3 RENAME]$\forall(z_1, z_2, z_3) :< AFAC3(z_1, z_2, z_3, n) = (z_1, z_2 + 1, z_3, n) >$

8. [$\forall$ Elim 7]$< AFAC3(x_1, x_2, x_3, n) = (x_1, x_2 + 1, x_3, n) >$

9. [= Sub 8, 6]$< GFAC1(x_1, x_2, x_3, n) = AFAC2(x_1, x_2 + 1, x_3, n) >$

10. [A2 RENAME]$\forall(z_1, z_2, z_3) :< AFAC2(z_1, z_2, z_3, n) = (z_1 * z_2, z_2, z_3, n) >$

11. [$\forall$ Elim 10]$< AFAC2(x_1, x_2 + 1, x_3, n) = (x_1 * (x_2 + 1), x_2 + 1, x_3, n) >$

12. [= Sub 11, 9]$< GFAC1(x_1, x_2, x_3, n) = (x_1 * (x_2 + 1), x_2 + 1, x_3, n) >$

13. [= Sub 3, 12]$< (y_1, y_2, y_3, n) = (x_1 * (x_2 + 1), x_2 + 1, x_3, n) >$

14. [= Elim 13]$< y_1 = x_1 * (x_2 + 1) >$

15. [= Elim 13]$< y_2 = x_2 + 1 >$

16. [Number Theory]$\forall(z_1, z_2) :< z_1 = fac(z_2) > \longrightarrow < z_1 * (z_2 + 1) = fac(z_2 + 1) >$

17. [$\forall$ Elim 16]$< x_1 = fac(x_2) > \longrightarrow < x_1 * (x_2 + 1) = fac(x_2 + 1) >$

18. [$\longrightarrow$ Elim 4, 17]$< x_1 * (x_2 + 1) = fac(x_2 + 1) >$

19. [= Sub 14, 18]$< y_1 = fac(x_2 + 1) >$

20. [= Sub 15, 19]$< y_1 = fac(y_2) >$

21. [$\longrightarrow$ Intro 2, 3-20]$(< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >$

22.[$\forall$ Intro 1, 2-21]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) : (< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >$

Proof 4 :

The meaning of the next proof is clear : If we apply $GFAC1$ 0 times. nothing happens to the output. Proving [D4] :
$\forall(x_1, x_2, x_3) :< WFAC1A(x_1, x_2, x_3, n, 0) = (x_1, x_2, x_3, n) >$

Proof 4:

1. [W1a1 RENAME]$\forall(z_1, z_2, z_3, z_4) :< (z_4 = 0) >$
   $\longrightarrow< WFAC1A(z_1, z_2, z_3, n, z_4) = (z_1, z_2, z_3, n) >$
2. [$\forall$ Elim 1]$\forall(z_1, z_2, z_3) :< (0 = 0) >$
   $\longrightarrow< WFAC1A(z_1, z_2, z_3, n, 0) = (z_1, z_2, z_3, n) >$
3. [Number Theory]$< 0 = 0 >$
4. [$\longrightarrow$ Elim 3,2]$\forall(z_1, z_2, z_3) :< WFAC1A(z_1, z_2, z_3, n, 0) = (z_1, z_2, z_3, n) >$

Proof 5 :

Proving [D5] :

$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, 0) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow< y_1 = fac(y_2) >$

Proof 5:

1. [Variables]$(y_1, y_2, y_3, x_1, x_2, x_3)$
2. [Assumption]$< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, 0) > \wedge < x_1 = fac(x_2) >$
3. [D4 RENAME]$\forall(z_1, z_2, z_3) :< WFAC1A(z_1, z_2, z_3, n, 0) = (z_1, z_2, z_3, n) >$
4. [$\forall$ Elim 3]$< WFAC1A(x_1, x_2, x_3, n, 0) = (x_1, x_2, x_3, n) >$
5. [$\wedge$ Elim 2]$< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, 0) >$
6. [=Sub 5,4]$< (y_1, y_2, y_3, n) = (x_1, x_2, x_3, n) >$
7. [=Elim 6]$< y_1 = x_1 >$
8. [=Elim 6]$< y_2 = x_2 >$
9. [$\wedge$ Elim 2]$< x_1 = fac(x_2) >$
10. [=Sub 9,7]$< y_1 = fac(x_2) >$
11. [=Sub 10,8]$< y_1 = fac(y_2) >$
12. [$\longrightarrow$ Intro 2, 3-11]$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, 0) >$
    $\wedge < x_1 = fac(x_2) >) \longrightarrow< y_1 = fac(y_2) >$
13. [$\forall$ Intro 1, 2-12]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) : ((< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, 0) >$
    $\wedge < x_1 = fac(x_2) >) \longrightarrow< y_1 = fac(y_2) >)$

Proof 6 :

Proof 5 was a proof of the base case of the factorial property that we are trying to demonstrate is satisfied by $WFAC1$. This is the general case. Proving [D6] :

$(\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$((< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind - 1) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow < y_1 = fac(y_2) >))$
$\longrightarrow$
$(\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$((< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow < y_1 = fac(y_2) >))$

Proof 6:

1. [Assumption] $\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
   $(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind - 1) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >$
2. [Variables] $(z_1, z_2, z_3, w_1, w_2, w_3)$
3. [Assumption] $(< (z_1, z_2, z_3, n) = WFAC1A(w_1, w_2, w_3, n, ind) > \wedge < w_1 = fac(w_2) >)$
4. [Number Theory] $(\forall(x) : (< 0 < x > \longrightarrow (< 0 = x - 1 > \vee < 0 < x - 1 >)))$
5. [$\forall$ Elim 4] $(< 0 < ind > \longrightarrow (< 0 = ind - 1 > \vee < 0 < ind - 1 >))$
6. [Induction 1] $< 0 < ind >$
7. [$\longrightarrow$ Elim 6, 5] $< 0 = ind - 1 > \vee < 0 < ind - 1 >$
8. [And Elim 3] $< (z_1, z_2, z_3, n) = WFAC1A(w_1, w_2, w_3, n, ind) >$
9. [W1a2] $\forall(y_1, y_2, y_3, y_4)$:
   $(< 0 < y_4 > \longrightarrow < WFAC1A(y_1, y_2, y_3, n, y_4) = GFAC1(WFAC1A(y_1, y_2, y_3, n, y_4 - 1)) >)$
10. [$\forall$ Elim 9] $(< 0 < ind > \longrightarrow$
    $< WFAC1A(w_1, w_2, w_3, n, ind) = GFAC1(WFAC1A(w_1, w_2, w_3, n, ind - 1)) >)$
11. [$\longrightarrow$ Elim 6, 10]
    $(< WFAC1A(w_1, w_2, w_3, n, ind) = GFAC1(WFAC1A(w_1, w_2, w_3, n, ind - 1)) >)$

12. [Assumption] $< 0 = ind - 1 >$
13. [= Elim 11,12] $< WFAC1A(w_1, w_2, w_3, n, ind) = GFAC1(WFAC1A(w_1, w_2, w_3, n, 0)) >$
14. [D4] $\forall(y_1, y_2, y_3) :< WFAC1A(y_1, y_2, y_3, n, 0) = (y_1, y_2, y_3, n) >$
15. [$\forall$ Elim 14] $< WFAC1A(w_1, w_2, w_3, n, 0) = (w_1, w_2, w_3, n) >$
16. [=Elim 15, 13] $< WFAC1A(w_1, w_2, w_3, n, ind) = GFAC1(w_1, w_2, w_3, n) >$
17. [And Elim 3] $< (z_1, z_2, z_3, n) = WFAC1A(w_1, w_2, w_3, n, ind) >$
18. [=Elim 17, 16] $< (z_1, z_2, z_3, n) = GFAC1(w_1, w_2, w_3, n) >$
19. [And Elim 3] $< w_1 = fac(w_2) >$
20. [And Intro 18, 19] $(< (z_1, z_2, z_3, n) = GFAC1(w_1, w_2, w_3, n) > \wedge < w_1 = fac(w_2) >)$
21. [D3] $\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
    $(< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >$
22. [$\forall$ Elim 21] $(< (z_1, z_2, z_3, n) = GFAC1(w_1, w_2, w_3, n) > \wedge < w_1 = fac(w_2) >)$
    $\longrightarrow < z_1 = fac(z_2) >$

500. [⟶ Elim 20, 22]$< z_1 = fac(z_2) >$

501. [Assumption]$< 0 < ind - 1 >$
502. [W1a3]$\forall(x_1, x_2, x_3, x_4) : \exists(v_1, v_2, v_3) : (v_1, v_2, v_3, n) = WFAC1A(x_1, x_2, x_3, n, x_4)$
503. [∀ Elim 502]$\exists(v_1, v_2, v_3) : (v_1, v_2, v_3, n) = WFAC1A(w_1, w_2, w_3, n, (ind - 1))$
504. [Variables]$(v_1^0, v_2^0, v_3^0) \quad (v_1^0, v_2^0, v_3^0, n) = WFAC1A(w_1, w_2, w_3, n, (ind - 1))$
505. [∀ Elim 1]$(< (v_1^0, v_2^0, v_3^0, n) = WFAC1A(w_1, w_2, w_3, n, (ind - 1)) >$
$\wedge < w_1 = fac(w_2) >) \longrightarrow < v_1^0 = fac(v_2^0) >$
506. [And Elim 3]$< w_1 = fac(w_2) >$
507. [And Intro 504, 506]$< (v_1^0, v_2^0, v_3^0, n) = WFAC1A(w_1, w_2, w_3, n, (ind - 1)) >$
$\wedge < w_1 = fac(w_2) >$
508. [⟶ Elim 507, 505]$< v_1^0 = fac(v_2^0) >$
509. [=Sub 504, 11]$< WFAC1A(w_1, w_2, w_3, n, ind) = GFAC1(v_1^0, v_2^0, v_3^0, n) >$
510. [=Sub 8, 509]$< (z_1, z_2, z_3, n) = GFAC1(v_1^0, v_2^0, v_3^0, n) >$
511. [D3]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$(< (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow < y_1 = fac(y_2) >$
512. [∀ Elim 511]$(< (z_1, z_2, z_3, n) = GFAC1(v_1^0, v_2^0, v_3^0, n) > \wedge < v_1^0 = fac(v_2^0) >)$
$\longrightarrow < z_1 = fac(z_2) >$
513. [∧ Intro 510, 508]$< (z_1, z_2, z_3, n) = GFAC1(v_1^0, v_2^0, v_3^0, n) > \wedge < v_1^0 = fac(v_2^0) >$
994. [⟶ Elim 512, 513]$< z_1 = fac(z_2) >$
995. [∃ Elim 503, 504-994]$< z_1 = fac(z_2) >$
996. [∨ Elim 7, 12-500, 501-995]$< z_1 = fac(z_2) >$
997. [⟶ Intro 3, 4-996]
$(< (z_1, z_2, z_3, n) = WFAC1A(w_1, w_2, w_3, n, ind) > \wedge < w_1 = fac(w_2) >)$
$\longrightarrow < z_1 = fac(z_2) >$
998. [∀ Intro 2-997]$\forall(z_1, z_2, z_3, w_1, w_2, w_3) :$
$(< (z_1, z_2, z_3, n) = WFAC1A(w_1, w_2, w_3, n, ind) > \wedge < w_1 = fac(w_2) >) \longrightarrow < z_1 = fac(z_2) >$
999. [998 Rename]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >$
1000. [⟶ 1, 2-999]
$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind - 1) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >)$
$\longrightarrow$
$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind) > \wedge < x_1 = fac(x_2) >) \longrightarrow < y_1 = fac(y_2) >)$

The other S_L calculus rules are now introduced.

- Collapse rule :

$$\frac{\#1, ([D_1|A_1][, [D_p|A_p]]*'), \#2}{\#1, [D_1|A_1][, [D_p|A_p]]*', \#2} \text{ Collapse}$$

The collapse rule allows us to derive something of the form : $FUNC1(FOO(x, y), n) = FUNC1(a, b, n)$ from $FOO(x, y) = (a, b)$. If we didn't have such a rule, we would be stuck at $FUNC1(FOO(x, y), n) = ((a, b), n)$ with no way to "collapse" the inner parenthesis.

- Induction rules

$$\frac{}{< 0 < ind >} \text{ ind}$$

$$\frac{F\{x => 0\} \qquad F\{x => (ind - 1)\} \longrightarrow F\{x => ind\}}{\forall(x) : F} \text{ Induction}$$

Proof 7 :

We can now show an example of the use of induction in a derivation. We have proved the base case in [D5] and the general case in [D6]. [D7] now completes the picture. Notice how the lemmas that were proved before are reused. Proving [D7] :

$\forall(w, y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, w) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow (< y_1 = fac(y_2) >)$

Proof 7:

1.[D5]$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, 0) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow (< y_1 = fac(y_2) >)$

2.[D6]
$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind - 1) > \wedge < x_1 = fac(x_2) >) \longrightarrow (< y_1 = fac(y_2) >)$
$\longrightarrow$
$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, ind) > \wedge < x_1 = fac(x_2) >) \longrightarrow (< y_1 = fac(y_2) >)$

3.[Induction 1, 2]$\forall(w) : \forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$((< (y_1, y_2, y_3, n) = WFAC1A(x_1, x_2, x_3, n, w) > \wedge < x_1 = fac(x_2) >)$
$\longrightarrow (< y_1 = fac(y_2) >)$

4.4 S_L theories

Definition 4.4.1 An S_L *theory* M is a triple $M = \{\Lambda_M, \Gamma_M, \Upsilon_M, \}$ where :

1. Λ_M is a finite set of variables
2. Γ_M is a finite set of well-formed S_L expressions such that for any $\gamma \in \Gamma_M$, if the variable x occurs in γ, then either x is bound in γ or $x \in \Lambda_M$
3. Υ_M is the set of well-formed expressions defined as :

$$\Upsilon_M = \{\gamma | \Gamma_M \vdash \gamma\}$$

We often call S_L theories "models". This is only syntax and is not to be confused with the usual notion of a model in logic.

Definition 4.4.2 For any model M, we write $M \vdash \gamma$ if $\gamma \in \Upsilon_M$.

4.5 Number Theory Axioms

In the previous sections, some lines in some example proofs were justified by the label '[Number Theory]'. This is our way of introducing sentences that

codify true properties of natural numbers. Typically, these properties would be derived from a set of axioms. One such possible set is proposed below.

Remark 4.5.1 There doesn't exist a finite set of axioms from which all the properties of **N** can be derived ([12] ch.6). This is one of the corollaries of Gödel's Incompleteness Theorem. Therefore, the set we propose is incomplete. It is an adaptation to S_L logic of the set that can be found in ([12], pg124), which is shown to be complete for a "*quite general set of sentences*"([12], pg124).

1. $\forall(x) : \neg(< x = 0 >)$
2. $\forall(x, y) :< (x + 1) = (y + 1) > \longrightarrow < x = y >$
3. $\forall(x) : (< x = 0 >) \vee (\exists(y) :< (y + 1) = x >)$
4. $\forall(x) :< x + 0 = x >$
5. $\forall(x, y) :< x + (y + 1) = (x + y) + 1 >$
6. $\forall(x) :< x * 0 = 0 >$
7. $\forall(x, y) :< x * (y + 1) = (x * y) + x >$
8. $\forall(x) :< exp(x, 0) = 1 >$
9. $\forall(x, y) :< exp(x, (y + 1)) = exp(x, y) * x >$
10. $\forall(x) :< x < x + 1 >$
11. $\forall(x, y) : (< x < y >) \longrightarrow ((< x + 1 < y >) \vee (< x + 1 = y >))$
12. $\forall(x, y) : \neg(< x < y >) \longrightarrow ((< x = y >) \vee (< y < x >))$
13. $\forall(x, y) : ((< x = y >) \vee (y < x)) \longrightarrow \neg(< x < y >)$
14. $\forall(x, y, z) : ((< x < y >) \wedge (< y < z >)) \longrightarrow (< x < z >)$

We also add to the set :

1. $< fac(0) = 1 >$

2. $\forall(x) :< fac(x+1) = fac(x) * (x+1) >$

Where fac is a new unary symbol. We do this to remain consistent with the examples that are provided where we use this symbol. We would expect proof designers to define their own set of axioms depending on what they need to prove (they would at least need to add the symbols that correspond to the properties they wish to build a related proof of, as well as some defining terms for these symbols). We would also urge to be extremely careful when adding axioms for the purpose of simplifying the proofs, as it is easy to introduce errors into the proofs in this way.

Chapter 5

Building S_L Theories from C^c Code Segments

In this chapter, the SMP algorithm is presented and justified. SMP is a recursive algorithm that builds S_L theories using C^c code segments as input.

Remark 5.0.2 We will also use the term "model" in an informal setting to denote an S_L theory.

5.1 Algorithm

5.1.1 Overview

Definitions and Notation

In the following algorithm description, P is a generic code segment that declares and uses only local variables $x_1, x_2, \ldots, x_p$, takes $n_1, n_2, \ldots, n_q$ as parameters. The model of P constructed by the SMP algorithm is named M_P. M_P is an S_L model whose set of sentences Γ_{M_P} describes the classes of functions (Tuple-Property classes) that the statements of P define. To build M_P, SMP adds sentences to Γ_{M_P} and variables to Λ_{M_P}. Γ_{M_P} is an empty set when the algorithm starts, while Λ_{M_P} contains P's parameters which are used as constants. Since P is itself a C^c statement (see chapter 3), SMP also adds to Γ_{M_P} the S_L definition of a Tuple-Property class to which the output

of a function equivalent to P corresponds. The name of this Tuple-Property class is the output of SMP.

We now give the definition of Code-Equivalency, which will be used in the justification of the algorithm processing choices. In the definitions proofs and justifications that appear in the chapter, we make extensive use of the notion of *logical variables*. In the context of Hoare Triple Logic, these are variables that do not occur in the actual program, but only in the logical expressions. We often give them values before or during the execution of the program, and they keep these values from then on.

Definition 5.1.1 (Code-Equivalent functions) Let $FUNC$ be a C^c statement of a program P in which the only local variables that are used (and declared) are $(v_1, \ldots, v_p)$. Let $f : \mathbf{N}^p \Longrightarrow \mathbf{N}^p$ be a recursive function. Let $(y_1, \ldots, y_p)$ be a set of logical variables that do not appear anywhere in P and chosen such that $(y_1, \ldots, y_p) = f(v_1, \ldots, v_p)$ holds just before $FUNC$s execution. If there is a derivation for the following Hoare Triple ,

$$\frac{|(y_1, \ldots, y_p) = f(v_1, \ldots, v_p)| \quad \text{PRE CONDITION}}{\texttt{FUNC}} \qquad \frac{|(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p)| \quad \text{POST CONDITION}}{}$$

we then say that $FUNC$ and f are *code-equivalent.*

Remark 5.1.2 Both input and output variables are lumped together in $(v_1, \ldots, v_p)$.

Remark 5.1.3 We note that every function $f : \mathbf{N}^n \Longrightarrow \mathbf{N}^m$ can be encoded as a function in $\mathbf{N}^p \Longrightarrow \mathbf{N}^p$ where $p \geq n, m$.

Remark 5.1.4 We notice that every recursive function from $\mathbf{N}^P$ to $\mathbf{N}^P$ also defines a Tuple-Property class. We will also speak of *Code-Equivalent Tuple-Property Classes*

Remark 5.1.5 In chapter 1, section 1.2.3, we gave a more intuitive version of this definition. In this context, the version we gave translates as:

$\|(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p)\|$	PRE CONDITION

FUNC

$\|(v_1, \ldots, v_p) = f(y_1, \ldots, y_p)\|$	POST CONDITION

We notice that the two versions are the same by the following :

$\|(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p) \wedge ((y_1^0, \ldots, y_p^0) = f(v_1, \ldots, v_p))\|$	PRE CONDITION
$\|(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p) \wedge ((y_1^0, \ldots, y_p^0) = f(y_1, \ldots, y_p))$	IMPLIED

FUNC

$\|(y_1^0 = v_1) \wedge \ldots \wedge (y_p^0 = v_p)$ (from code equivalence of f and PRE CONDITION)	
$\wedge((y_1^0, \ldots, y_p^0) = f(y_1, \ldots, y_p))\|$	IMPLIED
$\|(v_1, \ldots, v_p) = f(y_1, \ldots, y_p)\|$	IMPLIED

Which shows that the second 'official' definition implies the first, and the simple derivation that shows that the first 'non official definition' implies the second (simple when keeping in mind that f is from $\mathbb{N}^p$ to $\mathbb{N}^p$).

Lemma 5.1.6 (Code-Equivalency Composition) *Let $FUNC1$ and $FUNC2$ be two C^c statements of a program P in which the only local variables that are used (and declared) are $(v_1, \ldots, v_p)$. If f_1, $f_2 : \mathbb{N}^p \Longrightarrow \mathbb{N}^p$ are two functions Code-Equivalent to $FUNC1$ and to $FUNC2$ respectively, then the function defined by composition as :*

$$f_{1,2} = f_2 \circ f_1 : \mathbb{N}^p \Longrightarrow \mathbb{N}^p$$

is Code-Equivalent to the statement $[FUNC] = [FUNC1][FUNC2]$, the statement constructed from $FUNC1$ followed immediately by $FUNC2$.

Proof. *P is in the form :*

```
P(...)
{
.
.
    [FUNC1]
    [FUNC2]
.
.
}
```

Let $(y_1, \ldots, y_p)$, $(y_1^1, \ldots, y_p^1)$, $(y_1^2, \ldots, y_p^2)$ *be 3 tuples of logical variables that do not appear anywhere in* P *and chosen such that* $|(y_1, \ldots, y_p) = f_{1,2}(v_1, \ldots, v_p)|$, $|(y_1^1, \ldots, y_p^1) = f_1(v_1, \ldots, v_p)|$ *and* $|(y_1^2, \ldots, y_p^2) = f_2(v_1, \ldots, v_p)|$ *hold respectively just before the execution of* $FUNC$. $FUNC1$ *and* $FUNC2$. *We then construct the following Hoare Triple proof :*

$\|(y_1, \ldots, y_p) = f_{1,2}(v_1, \ldots, v_p)$ $\wedge (y_1^1, \ldots, y_p^1) = f_1(v_1, \ldots, v_p)\|$	*PRE CONDITION*
$\|(y_1, \ldots, y_p) = f_2 \circ f_1(v_1, \ldots, v_p)$ $\wedge (y_1^1, \ldots, y_p^1) = f_1(v_1, \ldots, v_p)\|$	*IMPLIED*
$\|(y_1, \ldots, y_p) = f_2(y_1^1, \ldots, y_p^1)$ $\wedge (y_1^1, \ldots, y_p^1) = f_1(v_1, \ldots, v_p)\|$	*IMPLIED*

FUNC1

$\|(y_1,\ldots,y_p) = f_2(y_1^1,\ldots,y_p^1)$ $\wedge(y_1^1 = v_1) \wedge \ldots \wedge (y_p^1 = v_p)\|$	*(f_1 Code-Equivalent to FUNC1)*
$\|(y_1,\ldots,y_p) = f_2(v_1,\ldots,v_p)\|$	*IMPLIED*
$\|(y_1,\ldots,y_p) = f_2(v_1,\ldots,v_p)$ $\wedge(y_1^2,\ldots,y_p^2) = f_2(v_1,\ldots,v_p)\|$	*IMPLIED*
$\|(y_1,\ldots,y_p) = (y_1^2,\ldots,y_p^2)\|$ $\wedge(y_1^2,\ldots,y_p^2) = f_2(v_1,\ldots,v_p)\|$	*IMPLIED*

FUNC2

$\|(y_1,\ldots,y_p) = (y_1^2,\ldots,y_p^2)$ $\wedge(y_1^2 = v_1) \wedge \ldots \wedge (y_p^2 = v_p)\|$	*(f_2 Code-Equivalent to FUNC2)*
$\|(y_1,\ldots,y_p) = (v_1,\ldots,v_p)\|$	*IMPLIED*
$\|(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p)\|$	*IMPLIED*

which satisfies the definition of Code-Equivalency for $f_{1,2}$ and FUNC. □

Lemma 5.1.7 *Let FUNC be a C^c statement of a program P in which the only local variables that are used (and declared) are $(v_1,\ldots.v_p)$. Let f be a code-equivalent function of FUNC. Let h be any recursive function from $\mathbf{N}^p$ to $\mathbf{N}$. Then, for every $(c_1,\ldots,c_p) \in \mathbf{N}^p$. if $h(f(c_1,\ldots.c_p)) = a$. there is a Hoare Triple derivation of :*

$\|(v_1 = c_1) \wedge \ldots \wedge (v_p = c_p)\|$	*PRE CONDITION*

FUNC

$\|h(v_1,\ldots,v_p) = a\|$	*POST CONDITION*

Before the Lemma is proven, a brief explanation as to why this is a key result is needed. Any function that outputs yes or no depending on whether or not a

tuple satisfies a given property must be recursive. Because the h in the Lemma can be any recursive function, the Lemma means that for any property that is satisfied by the possible outputs of a Code-Equivalent function on FUNC. the final state of FUNC's variables also satisfies that property.

Remark 5.1.8 The lemma is also used to establish a 'same level' soundness relationship between Hoare Triple Logic and the derivations that are made on models constructed by the SMP algorithm. This comes from the Code-Equivalency properties of the functions that are defined through SMP.

Proof.
Since f is code-equivalent to FUNC. there is an Hoare Triple proof of :

$\|(y_1,\ldots,y_p) = f(v_1,\ldots,v_p)\|$	*PRE CONDITION*

FUNC

$\|(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p)\|$	*POST CONDITION*

Since f is from $\mathbf{N}^p$ *to* $\mathbf{N}^p$ *it can be rewritten as :*

$$f(x_1,\ldots,x_p) = (f_1(x_1,\ldots,x_p),\ldots,f_p(x_1,\ldots,x_p))$$

for some recursive functions $f_1,\ldots,f_p$ *from* $\mathbf{N}^p$ *to* $\mathbf{N}$. *Therefore. there is a Hoare Triple derivation of:*

$\|(y_1,\ldots,y_p) = f(v_1,\ldots,v_p) \wedge (v_1 = c_1) \wedge \ldots \wedge (v_p = c_p)\|$	*PRE CONDITION*
$\|(y_1 = f_1(v_1,\ldots,v_p)) \wedge \ldots \wedge (y_p = f_p(v_1,\ldots,v_p))$ $\wedge(y_1,\ldots,y_p) = f(v_1,\ldots,v_p) \wedge (v_1 = c_1) \wedge \ldots \wedge (v_p = c_p)\|$	*IMPLIED*
$\|(y_1 = f_1(c_1,\ldots,c_p)) \wedge \ldots \wedge (y_p = f_p(c_1,\ldots,c_p))$ $\wedge(v_1 = c_1) \wedge \ldots \wedge (v_p = c_p) \wedge (y_1,\ldots,y_p) = f(v_1,\ldots,v_p)\|$	*IMPLIED*

FUNC

$\|(y_1 = f_1(c_1, \ldots, c_p)) \wedge \ldots \wedge (y_P = f_P(c_1, \ldots, c_p))$ $\wedge(y_1 = v_1) \wedge \ldots \wedge (y_p = v_p)\|$	*(By Code-Equivalence of f)*
$\|(v_1 = f_1(c_1, \ldots, c_p)) \wedge \ldots \wedge (v_P = f_P(c_1, \ldots, c_p))\|$	*IMPLIED*
$\|(v_1, \ldots, v_p) = f(c_1, \ldots, c_p)\|$	*IMPLIED*
$\|h(v_1, \ldots, v_p) = h(f(c_1, \ldots, c_p))\|$	*IMPLIED*
$\|h(v_1, \ldots, v_p) = a\|$	*POST CONDITION*

□

5.1.2 Algorithm's Description

General Description

SMP takes P as input and has access to M_P which it modifies during its execution. Initially, Γ_{M_P} is empty and Λ_{M_P} contains only the parameters of P : $n_1, n_2, \ldots, n_q$. SMP creates an S_L Tuple-Property Class definition F_1 that is code-equivalent* to the last statement of P and adds it to Γ_{M_P}. The algorithm is then recursively applied to P stripped of its last statement, and the result of the recursive call is composed with F_1. The return value is the name of a composition of functions defined in the model that codifies P in S_L.

Formal Description

SMP uses a primary finite set of bound variables names : $(v_1, \ldots, v_p)$, a secondary finite set of bound variables names : $(y_1, \ldots, y_p)$ and an infinite set of bound variables names : $(w_1, w_2, \ldots)$.

*Except for the while loop encoding, which creates a set of functions, one of which is Code-Equivalent to the loop

```
BEGIN SMP(CodeBlock P)
  If (IsEmpty(P))
    Return (v_1, ..., v_p, n_1, n_2, ..., n_q)
  Else
    Return Compose(Process(LastStatement(P)), SMP(MinusLastStatement(P)))
END SMP
```

```
BEGIN Process(CodeStatement S)
  If IsAssignment(S)
    Return ProcessAssignment(S)
  If IsIfStatement(S)
    Return ProcessIfStatement(S)
  If IsWhileLoop(S)
    Return ProcessWhileStatement(S)
END Process
```

The major part of the work is then done in the three Process functions. Note that we will not define formally such functions as *Compose*, *LastStatement*, *MinusLastStatement*, because their meaning is obvious.

5.1.3 ProcessAssignment

The algorithm builds base case definition expressions using the assignment statements and the variables of P, then uses these to build more complex function definitions following the structure of P. Given AS_j, a C^c assignment statement labeled [j] of the general form:

```
[j] x_[i] = F_j(x_1, ..., x_p, n_1, ..., n_q)
```

BEGIN ProcessAssignment(AS_j)

$\Gamma_M ::= \Gamma_M \cup$
$\{\forall(v_1,\ldots,v_p):$
$A[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q) = (v_1,\ldots,v_{i-1},F_j(v_1,\ldots,v_p,n_1,\ldots,n_q),$
$v_{i+1},\ldots,v_p,n_1,\ldots,n_q)\}$

$\Gamma_M ::= \Gamma_M \cup$
$\{\forall(v_1,\ldots,v_p):$
$\exists(y_1,\ldots,y_p):$
$(y_1,\ldots,y_p,n_1,\ldots,n_q) = A[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q)\}$

Return $A[P][j]$
END ProcessAssignment

Remark 5.1.9 The first sentence that is added to the model by ProcessAssignment is the definition of the function $A[P][j]$. Refer to section 5.1.6 for an explanation of the last sentence. All the process functions add a statement in this form.

Example :

Building model from :

```
Vars :x_1 x_2 x_3 x_4
Return Var :x_4
Constants : n_1 n_2

Poly( n_1,  n_2)
{
[4]x_1 = (n_1 + 1) * 5;
[3]x_2 = 3 * (n_2 + n_1);
[2]x_3 = x_1 + x_2;
[1]x_4 = x_2 + x_3;
}
```

model :Poly
Constants :$n_1 n_2$
$\forall(x_1,x_2,x_3,x_4) : (Poly(x_1,x_2,x_3,x_4,n_1,n_2) = (APoly1(APoly2(APoly3(APoly4(x_1,x_2,x_3,x_4,n_1,n_2))))$
Γ_M :
Defining : $APoly1$ from assignment statement :-$[1]x_4 = x_2 + x_3$;- $\forall(x_1,x_2,x_3,x_4) : (APoly1(x_1,x_2,x_3,x_4,n_1,n_2) = (x_1,x_2,x_3,(x_2+x_3),n_1,n_2))$ $\forall(x_1,x_2,x_3,x_4) : (\exists(y_1,y_2,y_3,y_4) : ((y_1,y_2,y_3,y_4) = APoly1(x_1,x_2,x_3,x_4,n_1,n_2)))$ End Definition ($APoly1$)
Defining : $APoly2$ from assignment statement :-$[2]x_3 = x_1 + x_2$;- $\forall(x_1,x_2,x_3,x_4) : (APoly2(x_1,x_2,x_3,x_4,n_1,n_2) = (x_1,x_2,(x_1+x_2),x_4,n_1,n_2))$ $\forall(x_1,x_2,x_3,x_4) : (\exists(y_1,y_2,y_3,y_4) : ((y_1,y_2,y_3,y_4) = APoly2(x_1,x_2,x_3,x_4,n_1,n_2)))$ End Definition ($APoly2$)
Defining : $APoly3$ from assignment statement :-$[3]x_2 = 3*(n_2+n_1)$;- $\forall(x_1,x_2,x_3,x_4) : (APoly3(x_1,x_2,x_3,x_4,n_1,n_2) = (x_1,(3*(n_2+n_1)),x_3,x_4,n_1,n_2))$ $\forall(x_1,x_2,x_3,x_4) : (\exists(y_1,y_2,y_3,y_4) : ((y_1,y_2,y_3,y_4) = APoly3(x_1,x_2,x_3,x_4,n_1,n_2)))$ End Definition ($APoly3$)
Defining : $APoly4$ from assignment statement :-$[4]x_1 = (n_1+1)*5$;- $\forall(x_1,x_2,x_3,x_4) : (APoly4(x_1,x_2,x_3,x_4,n_1,n_2) = (((n_1+1)*5),x_2,x_3,x_4,n_1,n_2))$ $\forall(x_1,x_2,x_3,x_4) : (\exists(y_1,y_2,y_3,y_4) : ((y_1,y_2,y_3,y_4) = APoly4(x_1,x_2,x_3,x_4,n_1,n_2)))$ End Definition ($APoly4$)

Justification for the SMP Assignment Encoding

Lemma 5.1.10 *Let AP_j be a function defined on an assignment statement S by the SMP algorithm. AP_j is Code-Equivalent to S.*

Proof.

We define logical variables $(y_1,\ldots,y_p)$ such that

$$(y_1,\ldots,y_p,n_1,\ldots,n_q) = AP_j(v_1,...,v_p,n_1,\ldots,n_q)$$

holds just before the execution of the statement S.

If S is an assignment statement, P is in the form :

```
P(n_1, ..., n_q)
{
v_1 = ...;
v_2 = ...;
.
.
v_p = ...;
.
.
 [j] v_[i] = F_j(v_1, ..., v_p, n_1, ..., n_q); //(S)
.
.
}
```

We can now derive :

$\|(y_1, \ldots, y_p, n_1, \ldots, n_q) = AP_j(v_1, \ldots, v_p, n_1, \ldots, n_q)\|$	*PRE CONDITION*
$\|(y_1 = v_1) \wedge \ldots \wedge (y_i = F_j(v_1, \ldots, v_p, n_1, \ldots, n_p)) \wedge \ldots (y_p = v_p)\|$ *(This follows from the definition of AP_j)*	*IMPLIED*

```
[j] v_[i] = F_j(v_1, ..., v_p, n_1, ..., n_q);
```

$\|(y_1 = v_1) \wedge \ldots \wedge (y_i = v_i) \wedge \ldots (y_p = v_p)\|$	*ASSIGNMENT*

The derivation satisfies the definition of Code-Equivalency of AP_j on S.

□

5.1.4 ProcessIf

Given a statement $IfStatement_j$ labeled j in the form:

```
If(COND)
{
        [If Part]
}
Else
{
        [Else Part]
}
```

BEGIN ProcessIf($IfStatement_j$)

Cond ::= ProcessCond(GetCond($IfStatement_j$))
IfFuncName ::= SMP(IfBodyPart($IfStatement_j$))
ElseFuncName ::= SMP(ElseBodyPart($IfStatement_j$))

$$\Gamma_M ::= \Gamma_M \cup \{\forall(v_1,\ldots,v_p) : (Cond) \longrightarrow I[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q) = IfFuncName(v_1,\ldots,v_p,n_1,\ldots,n_q)\}$$

$$\Gamma_M ::= \Gamma_M \cup \{\forall(v_1,\ldots,v_p) : \neg(Cond) \longrightarrow I[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q) = ElseFuncName(v_1,\ldots,v_p,n_1,\ldots,n_q)\}$$

$$\Gamma_M ::= \Gamma_M \cup \{\forall(v_1,\ldots,v_p) : \exists(y_1,\ldots,y_p) : (y_1,\ldots,y_p,n_1,\ldots,n_q) = I[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q)\}$$

Return $I[P][j]$

END ProcessIf

Remark 5.1.11 *ProcessCond(Conditionalexpression)* simply replaces the '==' C^c operator by the S_L = symbol (if it occurs in the expression), and

the program local variables symbols that occur in the expression by the appropriate quantified variables. The definition of *GetCond*(), *IfBodyPart*() and *ElseBodyPart*() are the expected ones.

Example:

Building model from :

```
Vars :x_1
Return Var :x_1
Constants : n_1 n_2 n_3

max( n_1, n_2, n_3)
{
    if(n_1 < n_2) {

        if(n_2 < n_3)
        {
            [4]x_1 = n_3;
        }
        else
        {
            [3]x_1 = n_2;
        }

    }
    else
    {
        if(n_1 < n_3)
        {
            [2]x_1 = n_3;
        }
        else
        {
            [1]x_1 = n_1;
```

```
        }
    }
}
```

model :max
Constants :$n_1 n_2 n_3$
$\forall(x_1) : (max(x_1, n_1, n_2, n_3) = (Imax3(x_1, n_1, n_2, n_3)))$
Γ_M :
Defining : $Amax4$ from assignment statement :-$[4]x_1 = n_3$;- $\forall(x_1) : (Amax4(x_1, n_1, n_2, n_3) = ((n_3), n_1, n_2, n_3))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Amax4(x_1, n_1, n_2, n_3)))$ End Definition ($Amax4$)
Defining : $Amax3$ from assignment statement :-$[3]x_1 = n_2$;- $\forall(x_1) : (Amax3(x_1, n_1, n_2, n_3) = ((n_2), n_1, n_2, n_3))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Amax3(x_1, n_1, n_2, n_3)))$ End Definition ($Amax3$)
Defining : $Imax1$ from if statement :-*if* $(n_2 < n_3)\{\ldots\}$ *else* $\{\ldots\}$- $\forall(x_1) : ((n_2 < n_3) \longrightarrow (Imax1(x_1, n_1, n_2, n_3) = (Amax4(x_1, n_1, n_2, n_3))))$ $\forall(x_1) : (\neg(n_2 < n_3) \longrightarrow (Imax1(x_1, n_1, n_2, n_3) = (Amax3(x_1, n_1, n_2, n_3))))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Imax1(x_1, n_1, n_2, n_3)))$ End Definition ($Imax1$)
Defining : $Amax2$ from assignment statement :-$[2]x_1 = n_3$;- $\forall(x_1) : (Amax2(x_1, n_1, n_2, n_3) = ((n_3), n_1, n_2, n_3))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Amax2(x_1, n_1, n_2, n_3)))$ End Definition ($Amax2$)

Defining : $Amax1$ from assignment statement :-$[1]x_1 = n_1$:- $\forall(x_1) : (Amax1(x_1, n_1, n_2, n_3) = ((n_1), n_1, n_2, n_3))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Amax1(x_1, n_1, n_2, n_3)))$ End Definition ($Amax1$)
Defining : $Imax2$ from if statement :-*if* $(n_1 < n_3)\{\ldots\}$ *else* $\{\ldots\}$- $\forall(x_1) : ((n_1 < n_3) \longrightarrow (Imax2(x_1, n_1, n_2, n_3) = (Amax2(x_1, n_1, n_2, n_3))))$ $\forall(x_1) : (\neg(n_1 < n_3) \longrightarrow (Imax2(x_1, n_1, n_2, n_3) = (Amax1(x_1, n_1, n_2, n_3))))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Imax2(x_1, n_1, n_2, n_3)))$ End Definition ($Imax2$)
Defining : $Imax3$ from if statement :-*if* $(n_1 < n_2)\{\ldots\}$ *else* $\{\ldots\}$- $\forall(x_1) : ((n_1 < n_2) \longrightarrow (Imax3(x_1, n_1, n_2, n_3) = (Imax1(x_1, n_1, n_2, n_3))))$ $\forall(x_1) : (\neg(n_1 < n_2) \longrightarrow (Imax3(x_1, n_1, n_2, n_3) = (Imax2(x_1, n_1, n_2, n_3))))$ $\forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = Imax3(x_1, n_1, n_2, n_3)))$ End Definition ($Imax3$)

Justification for the SMP If/Else Encoding

Lemma 5.1.12 *Let IP_j be a function defined on an if/else statement S by the SMP algorithm. IP_j is Code-Equivalent to S.*

Proof.

We define logical variables $(y_1, \ldots, y_p)$ such that

$$(y_1, \ldots, y_p, n_1, \ldots, n_q) = IP_j(v_1, ..., v_p, n_1, \ldots, n_q)$$

holds just before the execution of the statement S. we know that P is in the form :

```
P(n_1, ..., n_q)
{
v_1 = ...;
v_2 = ...;
.
```

```
.
v_p = ...;
.
.
    If(COND)    //(S)
    {
        [If Part]
    }
    Else
    {
        [Else Part]
    }

.
.
}
```

Where we assume that the functions $If_j(v_1, ..., v_p, n_1, \ldots, n_q)$ *and* $Else_j(v_1, ... v_p, n_1, \ldots, n_q)$ *defined by the SMP algorithm on the [If Part] and the [Else Part] statements satisfy Code-Equivalency.*
IP_j *is in the general form :*

$$(Cond) \longrightarrow IP_j(v_1, ... v_p, n_1, \ldots, n_q) = If_j(v_1, ... v_p, n_1, \ldots, n_q)$$

$$\neg(Cond) \longrightarrow IP_j(v_1, ..., v_p, n_1, \ldots, n_q) = Else_j(v_1, ... v_p, n_1, \ldots, n_q)$$

We then derive the following Hoare Triple Proof:

$\|(y_1, \ldots, y_p, n_1, \ldots, n_q) = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q)\|$	*PRE CONDITION*

```
If(COND) {
```

$\|(Cond)$ $\wedge((y_1,\ldots,y_p,n_1,\ldots,n_q) = IP_j(v_1,\ldots,v_p,n_1,\ldots,n_q))\|$	*IF STATEMENT*
$\|(Cond)$ $\wedge((y_1,\ldots,y_p,n_1,\ldots,n_q) = IP_j(v_1,\ldots,v_p,n_1,\ldots,n_q))\|$ $\wedge((Cond) \longrightarrow (IP_j(v_1,...,v_p,n_1,\ldots,n_q) = If_j(v_1,\ldots.n_q)))$ *(Follows from definition of IP_j, see above)*	*IMPLIED*
$\|(IP_j(v_1,...,v_p,n_1,\ldots,n_q) = If_j(v_1,...,v_p,n_1,\ldots,n_q))$ $\wedge((y_1,\ldots,y_p,n_1,\ldots,n_q) = IP_j(v_1,...,v_p,n_1,\ldots.n_q))\|$	*IMPLIED*
$\|((y_1,\ldots,y_p,n_1,\ldots,n_q) = If_j(v_1,...,v_p.n_1,\ldots.n_q))\|$	*IMPLIED*

[If Part]

$\|(y_1 = v_1) \wedge \ldots \wedge (y_i = v_i) \wedge \ldots (y_p = v_p)\|$	*(By Induction Hypothesis)*

}

Else {

$\|\neg(Cond)$ $\wedge(y_1,\ldots,y_p,n_1,\ldots.n_q) = IP_j(v_1,\ldots,v_p,n_1,\ldots.n_q)\|$	*IF STATEMENT*
$\|\neg(Cond)$ $\wedge(y_1,\ldots,y_p,n_1,\ldots,n_q) = IP_j(v_1,\ldots,v_p,n_1,\ldots,n_q)\|$ $\wedge(\neg(Cond) \longrightarrow IP_j(v_1,...,v_p,n_1,\ldots,n_q) = Else_j(v_1,\ldots.n_q))$	*IMPLIED*
$\|(IP_j(v_1,\ldots,v_p,n_1,\ldots.n_q) = Else_j(v_1,\ldots,v_p,n_1,\ldots,n_q))$ $\wedge(y_1,\ldots,y_p,n_1,\ldots,n_q) = IP_j(v_1,\ldots,v_p,n_1,\ldots.n_q)\|$	*IMPLIED*
$\|(y_1,\ldots.y_p,n_1,\ldots,n_q) = Else_j(v_1,...,v_p,n_1,\ldots,n_q)\|$	*IMPLIED*

[Else Part]

$\|y_1 = v_1 \wedge \ldots \wedge y_i = v_i \wedge \ldots \wedge y_p = v_p\|$	*(By Induction Hypothesis)*

}

$\lvert y_1 = v_1 \wedge \ldots \wedge y_i = v_i \wedge \ldots y_p = v_p \rvert$	*IF STATEMENT*

IP_j thus satisfies the definition of Code-Equivalency on S. □

5.1.5 ProcessWhile

Given a statement $WStatement_j$ labeled j in the form:

```
while(COND)
{
        [Body Part]
}
```

BEGIN ProcessWhile($WStatement_j$)

Cond ::= ProcessCond(GetCond($WStatement_j$))
BodyFuncName ::= SMP(GetBodyPart($WStatement_j$))

$$\Gamma_M ::= \Gamma_M \cup \{\forall(v_1,\ldots,v_p,w) : ((0 = w) \longrightarrow (W[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q,w) = (v_1,\ldots,v_p,n_1,\ldots,n_q)))$$

$$\Gamma_M ::= \Gamma_M \cup \{\forall(v_1,\ldots,v_p,w) : ((0 < w) \longrightarrow (W[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q,w) = (BodyFuncName(W[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q,w-1)))))$$

$$\Gamma_M ::= \Gamma_M \cup \{\forall(v_1,\ldots,v_p,w) : \exists(y_1,\ldots,y_p) : (y_1,\ldots,y_p,n_1,\ldots,n_q) = W[P][j](v_1,\ldots,v_p,n_1,\ldots,n_q,w)\}$$

END ProcessWhile

Example:

Building model from :

```
Vars :x_1 x_2 x_3
Constants : n_1 n_2
```

```
func (n_1, n_2){

            [6]x_1 = 0;
            [5]x_2 = 1;
            [4]x_3 = 1;

            while(x_1 < n_2){
                    [3]x_1 = x_1 + 1;
                    [2]x_2 = n_1 * x_2;
                    [1]x_3 = x_3 * x_1;
            }
}
```

model :func
Constants :$n_1 n_2$
$\forall(x_1, x_2, x_3) : (func(x_1, x_2, x_3, n_1, n_2) = (Wfunc1(Afunc4(Afunc5(Afunc6(x_1, x_2, x_3, n_1, n_2))))))$
Γ_M :
Defining : $Afunc1$ from assignment statement :-$[1]x_3 = x_3 * x_1$;- $\forall(x_1, x_2, x_3) : (Afunc1(x_1, x_2, x_3, n_1, n_2) = (x_1, x_2, (x_3 * x_1), n_1, n_2))$ $\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = Afunc1(x_1, x_2, x_3, n_1, n_2)))$ End Definition ($Afunc1$)
Defining : $Afunc2$ from assignment statement :-$[2]x_2 = n_1 * x_2$;- $\forall(x_1, x_2, x_3) : (Afunc2(x_1, x_2, x_3, n_1, n_2) = (x_1, (n_1 * x_2), x_3, n_1, n_2))$ $\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = Afunc2(x_1, x_2, x_3, n_1, n_2)))$ End Definition ($Afunc2$)
Defining : $Afunc3$ from assignment statement :-$[3]x_1 = x_1 + 1$;- $\forall(x_1, x_2, x_3) : (Afunc3(x_1, x_2, x_3, n_1, n_2) = ((x_1 + 1), x_2, x_3, n_1, n_2))$ $\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = Afunc3(x_1, x_2, x_3, n_1, n_2)))$ End Definition ($Afunc3$)

Defining : $Wfunc1$ from while statement :-*while*$(x_1 < n_2)\{\ldots\}$-

$\forall(x_1, x_2, x_3, w) : ((0 = w) \longrightarrow (Wfunc1(x_1, x_2, x_3, n_1, n_2, w) = (x_1, x_2, x_3, n_1, n_2)))$
$\forall(x_1, x_2, x_3, w) :$
$((0 < w)$
$\longrightarrow (Wfunc1(x_1, x_2, x_3, n_1, n_2, w) = Afunc1(Afunc2(Afunc3(Wfunc1(x_1, x_2, x_3, n_1, n_2, (w-1)))))))$

$\forall(x_1, x_2, x_3, w) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3) = Wfunc1(x_1, x_2, x_3, n_1, n_2, w)))$
End Definition ($Wfunc1$)

Defining : $Afunc4$ from assignment statement :-$[4]x_3 = 1$;-

$\forall(x_1, x_2, x_3) : (Afunc4(x_1, x_2, x_3, n_1, n_2) = (x_1, x_2, (1), n_1, n_2))$
$\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = Afunc4(x_1, x_2, x_3, n_1, n_2)))$

End Definition ($Afunc4$)

Defining : $Afunc5$ from assignment statement :-$[5]x_2 = 1$;-

$\forall(x_1, x_2, x_3) : (Afunc5(x_1, x_2, x_3, n_1, n_2) = (x_1, (1), x_3, n_1, n_2))$
$\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = Afunc5(x_1, x_2, x_3, n_1, n_2)))$

End Definition ($Afunc5$)

Defining : $Afunc6$ from assignment statement :-$[6]x_1 = 0$;-

$\forall(x_1, x_2, x_3) : (Afunc6(x_1, x_2, x_3, n_1, n_2) = ((0), x_2, x_3, n_1, n_2))$
$\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = Afunc6(x_1, x_2, x_3, n_1, n_2)))$

End Definition ($Afunc6$)

Justification for the While loops encoding

The first choice that we considered in attempting to code the looping structure in S_L logic was a recursive S_L statement which would have been roughly equivalent to rewriting the loop as the recursive function given below in pseudocode :

```
While(x_1, ..., x_p){

      if Not (Cond) Evaluate to (x_1, ..., x_p)
```

```
        else Evaluate to While(BodyFunc(x_1, ..., x_p))
}
```

Unfortunately , while this is a correct, complete and easily implementable encoding, it generates difficult proofs because the function is *Tail recursive.* This type of recursion does not create a stack when implemented since the result of the evaluation is entirely contained in the recursive call. Because of this, when proving properties by induction (and proofs on loops usually are by induction) about tail recursive functions it is very hard to establish a diminishing relationship (*measure*) between the variables. This means that it is very difficult to show that the above function actually terminates since the terminating condition is a combination of the processing that we do on the variables (*BodyFunc*) and the relationship that needs to eventually evaluate to 0 (*Cond*) In fact, when using ACL2 (more in chapter 6) on encodings of this kind, we have not been able to logically define the function (never mind proving something about it). ACL2 will reject it in its logic mode because it can not find a diminishing measure (although it will accept it in programming mode).

So instead, we chose a two part encoding. A looping is a multiple application of a function. Given a tuple $(x_1, x_2, \ldots, x_p)$ we know that the result of a finishing loop will be to apply a function $LOOP$ to $(x_1, x_2, \ldots, x_p)$, l times where $0 \leq l$. This can be informally written as $LOOP^l(x_1, x_2, \ldots, x_p)$.

We also know that applying the $LOOP$ function 0 times to $(x_1, x_2, \ldots, x_p)$ must yield $(x_1, x_2, \ldots, x_p)$. We can then construct the following expression :

$$LLOOP1(x_1, \ldots, x_p, 0) = (x_1 \ldots . x_p)$$

Since we also know that applying $LOOP$ to $(x_1, x_2, \ldots, x_p)$, n times is equivalent to applying $LOOP$ to $(x_1, x_2, \ldots, x_p)$ $n-1$ times and then applying $LOOP$ one more time to the result of that, we write :

$$LOOP^n(x_1, \ldots, x_p) = LOOP(LOOP^{n-1}(x_1, \ldots . x_p))$$

We define the recursive function $LOOP1$ as :

$$LLOOP1(x_1, \ldots, x_p, n) = LOOP(LLOOP1(x_1, \ldots . x_p, n-1))$$

Given a well formed definition of $LOOP$:

- $LLOOP1(x_1, \ldots, x_p, 0) = (x_1, \ldots, x_p)$
- $LLOOP1(x_1, \ldots, x_p, n) = LOOP(LLOOP1(x_1, \ldots, x_p, n-1))$

is the encoding of the looping function.

Lemma 5.1.13 *Let WP_j be a function defined on a while statement S of a program P by the SMP algorithm, where P is in the form :*

```
P(n_1, ..., n_q)
{
v_1 = ...;
v_2 = ...;
.
.
v_p = ...;
.
.
    while(COND)    //(S)
    {
        [While Body Part]
    }
.
.
}
```

and where we assume that the function $Body_j(v_1, \ldots, v_p, n_1, \ldots, n_q)$ defined by the SMP algorithm on the [While Body Part] statement satisfies Code-Equivalency.
WP_j is in the general form :

- $WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, 0) = (v_1, \ldots, v_p, n_1, \ldots, n_q)$
- $WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, w+1) = Body_j(WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, w))$

We now consider the set of functions from $\mathbf{N}^{p+q}$ to $\mathbf{N}^{p+q}$ generated by WP_j and defined as:

$FW = \{f_k | f_k(v_1, \ldots, v_p, n_1, \ldots, n_q) = WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, k), k \in \mathbf{N}\}$

If the while loop executes t times, then $f_t \in F_W$ is Code-Equivalent to S.
Justification. *Our justification is an induction on t.*

- *Base case : $t = 0$*

  We then derive the following Hoare Triple Proof:

$\|(y_1, \ldots, y_p, n_1, \ldots, n_q) = f_0(v_1, \ldots, v_p, n_1, \ldots, n_q)\|$	*PRE CONDITION*
$\|(y_1, \ldots, y_p, n_1, \ldots, n_q) = WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, 0)\|$	*IMPLIED*
$\|(y_1, \ldots, y_p, n_1, \ldots, n_q) = (v_1, \ldots, v_p, n_1, \ldots, n_q)\|$	*IMPLIED*

```
while(COND) {
.
[This  doesn't execute]
.
}
```

$\boxed{|y_1 = v_1 \wedge \ldots \wedge y_i = v_i \wedge \ldots \wedge y_p = v_p|\ \textit{IMPLIED}}$

f_0 thus satisfies the definition of Code-Equivalency on S.

Remark 5.1.14 We bypassed the traditional Hoare notation for the while loop in this case. We feel justified in doing so because we know by hypothesis that the while loop executes 0 times, and therefore does not have an impact on the state of the variables of P. The traditional Hoare Triple while rule on the other end makes no assumption as to the number of times the loop executes.

Remark 5.1.15 Hoare Triple notation is not particularly well adapted to these kind of proofs. In this case we are forced to take some liberties with the notation.

- *General Case : If f_t is Code-Equivalent to the S while loop executing t times, then f_{t+1} is Code-Equivalent to the S while loop executing $t + 1$ times.*

We first notice that :

$\|(y_1,\ldots,y_p,n_1,\ldots,n_q) = f_{t+1}(v_1,\ldots,v_p,n_1,\ldots,n_q)\|$	*PRE CONDITION*
$\|(y_1,\ldots,y_p,n_1,\ldots,n_q) = WP_j(v_1,\ldots,v_p,n_1,\ldots,n_q,t+1)\|$	*IMPLIED*
$\|(y_1,\ldots,y_p,n_1,\ldots,n_q) = Body_j(WP_j(v_1,\ldots,v_p,n_1,\ldots,n_q,t))\|$	*IMPLIED*
$\|(y_1,\ldots,y_p,n_1,\ldots,n_q) = Body_j(f_t(v_1,\ldots,v_p,n_1,\ldots,n_q))\|$	*IMPLIED*

is a valid derivation. We can consider a while loop executing $t+1$ times as a while loop executing t times followed immediately by one more execution of the body of the loop. Since by Induction Hypothesis. f_t is Code-Equivalent to the while loop executing t times. and since $Body_j$ is Code-Equivalent to the body of the while loop. then by the Code-Equivalency Composition lemma. the function defined as :

$$Body_j \circ f_t(v_1,\ldots,v_p,n_1,\ldots,n_q) = Body_j(f_t(v_1,\ldots,v_p,n_1,\ldots,n_q))$$

is Code-Equivalent to the while loop S executing $t+1$ times. This allows us to finish the proof with :

$\|(y_1,\ldots,y_p,n_1,\ldots,n_q) = f_{t+1}(v_1,\ldots,v_p,n_1,\ldots,n_q)\|$	*PRE CONDITION*
$\vdots$	
$\|(y_1,\ldots,y_p,n_1,\ldots,n_q) = Body_j(f_t(v_1,\ldots,v_p,n_1,\ldots,n_q))\|$	*IMPLIED*

```
while(cond)
{
//This executes t+1 times
}
```

$\|y_1 = v_1 \wedge \ldots \wedge y_i = v_i \wedge \ldots \wedge y_p = v_p\|$	*(By Code Equivalence of $Body_j \circ f_t$)*

Which satisfies the definition of Code-Equivalency for f_{t+1} on S.

□

Remark 5.1.16 While the lemma above does not show that the function defined by SMP on a while loop S is Code-Equivalent to it (it is not), it does show that any property that is proved of that function will be true of S. The reason for this is that the function that SMP defines on a while loop defines a set of functions, and given that the loop completes, then there is a function in that set that is Code-Equivalent to S. Therefore, any property that is true of the whole set must also be true of the Code-Equivalent function that is in the set.

5.1.6 Justification for the Last Sentence in the Process Functions

We still haven't explained the purpose of the last sentence that is added to the model by all the process functions. There is a slight problem with our function definitions. What if there is no tuple $(v_1, \ldots, v_p) \in \mathbf{N}^P$ such that

$$(v_1, \ldots, v_p, n_1, \ldots, n_q) = Func_j(x_1, \ldots, x_p, n_1, \ldots, n_q)$$

where $Func_j$ is a function defined through SMP? This problem can be reduced to the assignment functions encodings and could only happen if we were to come upon an assignment statement for which there are no $v_i \in \mathbf{N}$ such that

$$v_i = F_j(x_1, \ldots, x_p, n_1, \ldots, n_p)$$

for some possible values $(x_1, \ldots, x_p)$ of the local variables of the program before the statement's execution. However, this would also mean that the program would fail on execution in this case (run time error). The last sentences added to the model by each of the process functions allow us to ignore these possibilities. Future work will be concerned in eliminating these safeguard sentences by deriving them (possibly in a typed version of the S_L Logic). We note however that in the case of the C^c language, this does not matter, because C^c can only construct assignments using basic arithmetic operations $(+, -, *)$ that are safe in the sense that they are defined on all possible inputs in $\mathbf{N}^2$.

5.1.7 Complexity Analysis

We define the size of the input n to be the number of statements in the program that is the argument of the SMP algorithm. We count the number of recursive calls and the computational steps.

- Every 'if/else' statement produces 2 recursive calls and 1 extra computational step (3 steps).
- Every 'while' statement produces 1 recursive call and 1 additional computational step (2 steps).
- Every 'assignment' statement is counted as 1 step.
- There is one extra computational step for the processing of the whole program.

In the worst case, assume that there are n if/else statements. Since each if/else statement takes 3 steps, the total number of steps is $3n + 1$. Thus, the running time of the algorithm is $O(n)$: that is, linear time.

Chapter 6

Implementations and Results

6.1 The Nivea Implementation

Nivea is the name of the system that was built to illustrate the system that is outlined in this thesis. It is made of a series of tools that automate the creation of a Model of a program. The implementation of the Nivea system was done in perl.

The system contains 3 subdirectories :

- 'Nivea/Programs/' :
  The original files containing the programs that are entered into the Nivea system are stored in this directory. These programs can be written in either C^c or C language. A file can contain the code for several programs.

- 'Nivea/NormalizedPrograms/' :
  This directory contains the programs that have been normalized by the Nivea system using the 'NormalizeProgram' command.

- 'Nivea/Models/':
  This directory contains the Models of the Normalized programs whose Models were created using the 'Modelize' command.

6.2 Commands

6.2.1 'GetProgram FilePath [FileName]'

This command takes the file pointed to by 'FilePath' and verifies that all the programs that are in it are valid C or C^c programs. The file is then stored in the 'Nivea/Programs' directory, under the name specified by 'FileName'.

6.2.2 'NormalizeProgram FileName'

This command takes the file 'Nivea/Programs/FileName' as input and extracts the first complete C or C^c program that is in it. A file is then created in the 'Nivea/NormalizedPrograms' directory under the name of the program that was extracted from 'FileName'. The structure of the output file is :

```
Variables :x_1, ..., x_p
Return Var :x_r
Constants :n_1, ..., n_q

ProgName(n_1,  ..., n_q)
{
    [CODE]
}
```

where $x_1, \ldots, x_p$ are the renamed local variables of the program, $n_1 \ldots . n_q$ are the parameters that are passed to the program, x_r is the name of the return variable (if any) and [CODE] is the program itself where the parameters and local variables have been renamed according to above and the assignment statements have been labeled. For example, if the input file 'Nivea/Programs/Prog' contains :

```
int Fac(int n)
{
    int  res  = 1;
    int  ctr = 0;
```

```
    while(ctr < n)
    {
        ctr = ctr + 1;
        res = res * ctr;
    }

    return res;
}
```

The output file, 'Nivea/NormalizedPrograms/Fac' is :

```
Vars :x_1 x_2 x_3
Return Var :x_3
Constants :n_1

Fac(n_1)
{
[5]x_1  = 1;
[4]x_2 = 0;
while(x_2 < n_1)
{
[3]x_2 = x_2 + 1;
[2]x_1 = x_1 * x_2;
}
[1]x_3 = x_1;
}
```

Implementation:

Each line in the program is matched with the regular expression:

```
if($line =~ m/^(\s*)(\w+)(\s*)[^=]=[^=](.+?);$/){
```

which matches alphabetic characters followed by a single '=' character. If there is a match, then the line is a statement and the assigned variable is

compared with an array of local variables. If it is not in the array, it is added to it and a normalized local variable x_i is created and associated with it. It is assumed that every local variable is assigned at least once. The assignment statement counter is then incremented. Each parameter and local variables in each line of the program is then substituted by its normalized name, and the assignment statements of the program are numbered. starting with the last one.

6.2.3 'Modelize ProgName'

'Modelize' implements the SMP algorithm. 'Modelize' takes the file 'Nivea/NormalizedPrograms/ProgName' as input and outputs the file 'Nivea/Models/ProgName', which contains a model of 'ProgName'. The output is in latex2e code.

Implementation:

Some key parts of the perl code for the SMP and processes function can be found in the appendices, as well as some input/output layouts. In the ProcessWhile subroutine, the beginning of a possible implementation of a complete correctness while loop handling can also be found in comments. However insufficient testing does not allow us to include it into the thesis.

6.3 Experimental Results

In order to be able to compare our method with existing methods, we did a complete analysis of the example program *func* that was presented in chapter 1. we used Nivea to create the Model. The program *func* was defined as follows:

```
void func (int x, int n){

            int a = 0;
            int b = 1;
            int c = 1;

            while(n > a){
                    a = a + 1;
                    b = x * b;
                    c = c * a;
            }
}
```

After entering the program in the Nivea system and normalizing it. Nivea outputs:

```
Vars :x_1 x_2 x_3
Return Var :
Constants : n_1, n_2

func ( n_1, n_2)
{
[6]x_1 = 0;
[5]x_2 = 1;
[4]x_3 = 1;
while(x_1 < n_2)
{
[3]x_1 = x_1 + 1;
```

```
[2]x_2 = n_1 * x_2;
[1]x_3 = x_3 * x_1;
}
}
```

Where variables x, n, a, b, c have been substituted by n_1, n_2, x_1, x_2, x_3 respectively. Using the 'Modelize' command on *func*, we get the following Model definition :

Model :func
Constants :n_1, n_2
$[Mf]\forall(x_1, x_2, x_3) : func(x_1, x_2, x_3, n_1, n_2) = Wfunc1(Afunc4(Afunc5(Afunc6(x_1, x_2, x_3, n_1, n_2))), w)$
Γ_M :
Defining : $Afunc1$ from assignment statement :-[1]$x_3 = x_3 * x_1$:- $[Mf1]\forall(x_1, x_2, x_3) : (Afunc1(x_1, x_2, x_3, n_1, n_2) = (x_1, x_2, (x_3 * x_1), n_1, n_2))$ $[Mf2]\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3) = Afunc1(x_1, x_2, x_3, n_1, n_2)))$ End Definition ($Afunc1$)
Defining : $Afunc2$ from assignment statement :-[2]$x_2 = n_1 * x_2$:- $[Mf3]\forall(x_1, x_2, x_3) : (Afunc2(x_1, x_2, x_3, n_1, n_2) = (x_1, (n_1 * x_2), x_3, n_1, n_2))$ $[Mf4]\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3) = Afunc2(x_1, x_2, x_3, n_1, n_2)))$ End Definition ($Afunc2$)
Defining : $Afunc3$ from assignment statement :-[3]$x_1 = x_1 + 1$:- $[Mf5]\forall(x_1, x_2, x_3) : (Afunc3(x_1, x_2, x_3, n_1, n_2) = ((x_1 + 1), x_2, x_3, n_1, n_2))$ $[Mf6]\forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3) = Afunc3(x_1, x_2, x_3, n_1, n_2)))$ End Definition ($Afunc3$)

Defining : $Wfunc1$ from while statement :-*while*$(x_1 < n_2)\{\ldots\}$-

$[Mf7]\forall(x_1,x_2,x_3,w) : ((0 = w) \longrightarrow (Wfunc1(x_1,x_2,x_3,n_1,n_2,w) = (x_1,x_2,x_3,n_1,n_2)))$
$[Mf8]\forall(x_1,x_2,x_3,w) :$
$\quad ((0 < w) \longrightarrow$
$\quad\quad (Wfunc1(x_1,x_2,x_3,n_1,n_2,w) = Afunc1(Afunc2(Afunc3(Wfunc1(x_1,x_2,x_3,n_1,n_2,(w-1)))))))$
$[Mf9]\forall(x_1,x_2,x_3,w) : (\exists(y_1,y_2,y_3) : ((y_1,y_2,y_3,n_1,n_2) = Wfunc1(x_1,x_2,x_3,n_1,n_2,w)))$

End Definition ($Wfunc1$)

Defining : $Afunc4$ from assignment statement :-$[4]x_3 = 1$;-

$[Mf10]\forall(x_1,x_2,x_3) : (Afunc4(x_1,x_2,x_3,n_1,n_2) = (x_1,x_2,(1),n_1,n_2))$
$[Mf11]\forall(x_1,x_2,x_3) : (\exists(y_1,y_2,y_3) : ((y_1,y_2,y_3) = Afunc4(x_1,x_2,x_3,n_1,n_2)))$

End Definition ($Afunc4$)

Defining : $Afunc5$ from assignment statement :-$[5]x_2 = 1$:-

$[Mf12]\forall(x_1,x_2,x_3) : (Afunc5(x_1,x_2,x_3,n_1,n_2) = (x_1,(1),x_3,n_1,n_2))$
$[Mf13]\forall(x_1,x_2,x_3) : (\exists(y_1,y_2,y_3) : ((y_1,y_2,y_3) = Afunc5(x_1,x_2,x_3,n_1,n_2)))$

End Definition ($Afunc5$)

Defining : $Afunc6$ from assignment statement :-$[6]x_1 = 0$:-

$[Mf14]\forall(x_1,x_2,x_3) : (Afunc6(x_1,x_2,x_3,n_1,n_2) = ((0),x_2,x_3,n_1,n_2))$
$[Mf15]\forall(x_1,x_2,x_3) : (\exists(y_1,y_2,y_3) : ((y_1,y_2,y_3) = Afunc6(x_1,x_2,x_3,n_1,n_2)))$

End Definition ($Afunc6$)

Remark 6.3.1 In the definition of $func$, we added w as an unquantified variable. It is just a place holder that is used to keep the arity of $Wfunc1$ consistent. It could be read as "some w". We could also have quantified it with $\exists$, but we want to keep this sentence in the form of a definition (only $\forall$ quantifiers). Note however that the formal:

$\forall(x_1,x_2,x_3) : \exists(w) :$
$func(x_1,x_2,x_3,n_1,n_2) = Wfunc1(Afunc4(Afunc5(Afunc6(x_1,x_2,x_3,n_1,n_2))),w)$

only complicates proofs, without adding anything to the meaning of the sentence. Note also that it can be read as an assumption as to the termination of $func$.

Proof 1 :

We then set about to prove some properties of the $func$ function. We start by showing that the composition of functions that codifies the body of the while loop preserves the exponential and factorial relationships of its inputs on its outputs. This property is expressed by [D1]:

$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2))) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$

Proof of [D1]:

1.[Variables]$(y_1, y_2, y_3, x_1, x_2, x_3)$
2. [Assumption]$(< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2))) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
3. [$\wedge$ Elim 2]$< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2))) >$
4. [$\wedge$ Elim 2]$(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >)$
5. [Mf5]
6. [$\forall$ Elim 5]$< Afunc3(x_1, x_2, x_3, n_1, n_2) = ((x_1 + 1), x_2, x_3, n_1, n_2) >$
7. [= Sub 6, 3]$< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2((x_1 + 1), x_2, x_3, n_1, n_2)) >$
8. [Mf3]
9. [$\forall$ Elim 8]$< Afunc2((x_1 + 1), x_2, x_3, n_1, n_2) = ((x_1 + 1), (n_1 * x_2), x_3, n_1, n_2) >$
10. [= Sub 9, 7]$< (y_1, y_2, y_3, n_1, n_2) = Afunc1((x_1 + 1), (n_1 * x_2), x_3, n_1, n_2)$
11. [Mf1]
12. [$\forall$ Elim 11]$< Afunc1((x_1 + 1), (n_1 * x_2), x_3, n_1, n_2)$
$= ((x_1 + 1), (n_1 * x_2), (x_3 * (x_1 + 1)), n_1, n_2) >$
13. [= Sub 12, 10]$< (y_1, y_2, y_3, n_1, n_2) = ((x_1 + 1), (n_1 * x_2), (x_3 * (x_1 + 1)), n_1, n_2) >$

14. [= Elim 13]$< y_1 = x_1 + 1 >$
15. [= Elim 13]$< y_2 = n_1 * x_2 >$
16. [= Elim 13]$< y_3 = (x_3 * (x_1 + 1)) >$
17. [Number Theory]$\forall(z_1, z_2) :< z_1 = fac(z_2) >$
$\longrightarrow < z_1 * (z_2 + 1) = fac(z_2 + 1) >$
18. [$\forall$ Elim 17]$< x_3 = fac(x_1) > \longrightarrow < x_3 * (x_1 + 1) = fac(x_1 + 1) >$
19. [= Sub 16, 18]$< x_3 = fac(x_1) > \longrightarrow < y_3 = fac(x_1 + 1) >$
20. [= Sub 14, 19]$< x_3 = fac(x_1) > \longrightarrow < y_3 = fac(y_1) >$
21. [$\wedge$ Elim 4]$< x_3 = fac(x_1) >$
22. [$\longrightarrow$ Elim 20, 21]$< y_3 = fac(y_1) >$
23. [Number Theory]$\forall(z_1, z_2) :< z_1 = exp(n_1, z_2) >$
$\longrightarrow < n_1 * z_1 = exp(n_1, z_2 + 1) >$
24. [$\forall$ Elim 23]$< x_2 = exp(n_1, x_1) > \longrightarrow < n_1 * x_2 = exp(n_1, x_1 + 1) >$
25. [= Sub 14, 24]$< x_2 = exp(n_1, x_1) > \longrightarrow < n_1 * x_2 = exp(n_1, y_1) >$
26. [= Sub 15, 25]$< x_2 = exp(n_1, x_1) > \longrightarrow < y_2 = exp(n_1, y_1) >$
27. [$\wedge$ Elim 4]$< x_2 = exp(n_1, x_1) >$
28. [$\longrightarrow$ Elim 27, 26]$< y_2 = exp(n_1, y_1) >$
29. [$\wedge$ Intro 22, 28]$< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >$
30. [$\longrightarrow$ Intro 2, 3-29]$(< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$

27.[$\forall$ Intro 1, 2-30]
$\forall(y_1, y_2, y_3, x_1, x_2, x_3) : (< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$

Proof 2 :

We then prove some properties of the while loop itself. Setting up the base case for the induction, we start by showing that executing the loop 0 times has no effect on the variables. this is expressed in [D2] :
$\forall(x_1, x_2, x_3) :< Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) = (x_1, x_2, x_3, n_1, n_2) >$

Proof of [D2]:

1. [Mf7]
2. [$\forall$ Elim 1]$\forall(x_1, x_2, x_3) : (< 0 = 0 >)$
   $\longrightarrow (< Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) = (x_1, x_2, x_3, n_1, n_2) >)$
3. [Vars](z_1, z_2, z_3)
4. [$\forall$ Elim 2] $< (0 = 0) >$
   $\longrightarrow < Wfunc1(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) >$
5. [Number Theory]$< 0 = 0 >$
6. [$\longrightarrow$ Elim 4,5]$< Wfunc1(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) >$
7. [$\forall$ Intro 3, 4-6]$\forall(z_1, z_2, z_3)$:
   $< Wfunc1(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) >$

Proof 3 :

We now want to prove a property that comes from the while loop part of the code. Since the $Wfunc1$ function that was generated by the SMP algorithm doesn't code a function, but rather a countable set of functions. one of which is Code-Equivalent to the loop (see Justification for while loop encoding in chapter 5), proving a property that comes from the while loop requires us to prove that all the functions in the set generated by the encoded function have that property. This is done by induction. of which we now prove the base case. In the proof, we use [D2]. The base case of the induction is expressed as [D3] :

$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$

Proof of [D3]:

1.[Variables]$(y_1, y_2, y_3, x_1, x_2, x_3)$
2. [Assumption]$< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) >$
   $\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >)$
3. [D2 RENAME]$\forall(z_1, z_2, z_3) :< Wfunc1(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) >$
4. [$\forall$ Elim 3]$< Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) = (x_1, x_2, x_3, n_1, n_2) >$
5. [$\wedge$ Elim 2]$< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) >$
6. [=Sub 5,4]$< (y_1, y_2, y_3, n_1, n_2) = (x_1, x_2, x_3, n_1, n_2) >$
7. [=Elim 6]$< y_1 = x_1 >$
8. [=Elim 6]$< y_2 = x_2 >$

9. [=Elim 6]$< y_3 = x_3 >$
10. [$\wedge$ Elim 2]$(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >)$
11. [=Sub 10,7]$(< x_3 = fac(y_1) > \wedge < x_2 = exp(n, y_1) >)$
12. [=Sub 11,8]$(< x_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
13. [=Sub 12,9]$(< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
14. [$\longrightarrow$ Intro 2, 3-13]$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
15. [$\forall$ Intro 1, 2-14]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) : ((< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >))$

Proof 4 :

This is where we prove the general case of the while loop induction. we take some liberties with the line numbering to ease the reading of the proof. [D4] is expressed as:

$(\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$((< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind - 1) > \wedge(< x_3 =$
$fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)))$
$\longrightarrow$
$(\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$((< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind) > \wedge(< x_3 = fac(x_1) >$
$\wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)))$

Proof of [D4]:

1. [Assumption]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind - 1)$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$
2. [Variables]$(z_1, z_2, z_3, w_1, w_2, w_3)$
3. [Assumption]$(< (z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) >$
$\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >))$
4. [Number Theory]$(\forall(x) : (< 0 < x > \longrightarrow (< 0 = x - 1 > \vee < 0 < x - 1 >)))$

5. [$\forall$ Elim 4]($< 0 < ind > \longrightarrow (< 0 = ind - 1 > \vee < 0 < ind - 1 >)$)
6. [Induction 1]$< 0 < ind >$
7. [$\longrightarrow$ Elim 6, 5]$< 0 = ind - 1 > \vee < 0 < ind - 1 >$
8. [And Elim 3]$< (z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) >$
9. [Mf8]
10. [$\forall$ Elim 9]($< 0 < ind >$) $\longrightarrow$
    $(< Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) =$
    $Afunc1(Afunc2(Afunc3(Wfunc1(w_1, w_2, w_3, n_1, n_2, ind - 1))))) >)$
11. [$\longrightarrow$ Elim 6, 10]
    $(< Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) =$
    $Afunc1(Afunc2(Afunc3(Wfunc1(w_1, w_2, w_3, n_1, n_2, ind - 1)) >)$

12. [Assumption]$< 0 = ind - 1 >$
13. [= Elim 11,12]
    $< Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) =$
    $Afunc1(Afunc2(Afunc3(Wfunc1(w_1, w_2, w_3, n_1, n_2, 0))))) >$
14. [D2]$\forall(y_1, y_2, y_3) :< Wfunc1(y_1, y_2, y_3, n_1, n_2, 0) = (y_1, y_2, y_3, n_1, n_2) >$
15. [$\forall$ Elim 14]$< Wfunc1(w_1, w_2, w_3, n_1, n_2, 0) = (w_1, w_2, w_3, n_1, n_2) >$
16. [=Elim 15, 13]
    $< Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) = Afunc1(Afunc2(Afunc3(w_1, w_2, w_3, n_1, n_2) >$
17. [And Elim 3]$< (z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) >$
18. [=Elim 17, 16]$< (z_1, z_2, z_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(w_1, w_2, w_3, n_1, n_2) >$
19. [And Elim 3]($< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >$)
20. [And Intro 18, 19]
    $(< (z_1, z_2, z_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(w_1, w_2, w_3, n_1, n_2) >$
    $\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >))$
21. [D1]$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :$
    $(< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2) >$
    $\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
    $\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$
22. [$\forall$ Elim 21]$(< (z_1, z_2, z_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(w_1, w_2, w_3, n_1, n_2) >$
    $\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >))$
    $\longrightarrow (< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >)$
500. [$\longrightarrow$ Elim 20, 22]($< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >$)

501. [Assumption]$< 0 < ind - 1 >$
502. [Mf9]
503. [$\forall$ Elim 502]$\exists(v_1, v_2, v_3) : (v_1, v_2, v_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, (ind - 1))$
504. [Variables]$(v_1^0, v_2^0, v_3^0) \quad (v_1^0, v_2^0, v_3^0, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, (ind - 1))$

505. [∀ Elim 1]($< (v_1^0, v_2^0, v_3^0, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, (ind-1)) >$
$\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >)$
$\longrightarrow (< v_3^0 = fac(v_1^0) > \wedge < v_2^0 = exp(n_1, v_1^0) >)$

506. [And Elim 3]($< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >$)

507. [And Intro 504, 506]$< (v_1^0, v_2^0, v_3^0, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, (ind-1)) >$
$\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >)$

508. [$\longrightarrow$ Elim 507, 505]($< v_3^0 = fac(v_1^0) > \wedge < v_2^0 = exp(n_1, v_1^0) >$)

509. [=Sub 504, 11]
$< Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) = Afunc1(Afunc2(Afunc3(v_1^0, v_2^0, v_3^0, n_1, n_2) >$

510. [=Sub 8, 509]$< (z_1, z_2, z_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(v_1^0, v_2^0, v_3^0, n_1, n_2) >$

511. [D1]$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$

512. [∀ Elim 511]$(< (z_1, z_2, z_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(v_1^0, v_2^0, v_3^0, n_1, n_2) >$
$\wedge(< v_3^0 = fac(v_1^0) > \wedge < v_2^0 = exp(n_1, v_1^0) >))$
$\longrightarrow (< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >)$

513. [∧ Intro 510. 508]$< (z_1, z_2, z_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(v_1^0, v_2^0, v_3^0, n_1, n_2) >$
$\wedge(< v_3^0 = fac(v_1^0) > \wedge < v_2^0 = exp(n_1, v_1^0) >)$

994. [$\longrightarrow$ Elim 512, 513]($< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >$)

995. [∃ Elim 503, 504-994]($< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >$)

996. [∨ Elim 7, 12-500, 501-995]($< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >$)

997. [$\longrightarrow$ Intro 3, 4-996]$(< (z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) >$
$\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >))$
$\longrightarrow (< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >)$

998.[∀ Intro 2-997]$\forall(z_1, z_2, z_3, w_1, w_2, w_3)$:
$(< (z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) >$
$\wedge(< w_3 = fac(w_1) > \wedge < w_2 = exp(n_1, w_1) >))$
$\longrightarrow (< z_3 = fac(z_1) > \wedge < z_2 = exp(n_1, z_1) >)$

999.[998 Rename]$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)$

1000.[⟶ 1, 2-999]

$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind - 1) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >))$
$\longrightarrow$
$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >))$

Proof 5 :

Having proven both the base case and the general case, we are now in a position to prove that every function in the set of functions that is codified by the $Wfunc1$ function preserves the factorial and exponential properties. we prove this by induction, using the induction rule. We use both [D3] (base case) and [D4] (general case) to prove [D5] :

$\forall(w) : \forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, w) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$

Proof of [D5]:

1.[D3]$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfuncl(x_1, x_2, x_3, n_1, n_2, 0) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$

2.[D4]$(\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$((< (y_1, y_2, y_3, n_1, n_2) = Wfuncl(x_1, x_2, x_3, n_1, n_2, ind - 1) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)))$
$\longrightarrow$
$(\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$((< (y_1, y_2, y_3, n_1, n_2) = Wfuncl(x_1, x_2, x_3, n_1, n_2, ind) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n_1, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n_1, y_1) >)))$

3.[Induction 1, 2]$\forall(w) : \forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = Wfuncl(x_1, x_2, x_3, n_1, n_2, w) >$
$\wedge(< x_3 = fac(x_1) > \wedge < x_2 = exp(n, x_1) >))$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$

Proof 6:

That's it for the while loop. Now, we use [D5] to show the desired relationship on the $func$ function which codifies the program. [D6] is expressed as:
$\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$(< (y_1, y_2, y_3, n_1, n_2) = func(x_1, x_2, x_3, n_1, n_2) >$
$\longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
Note the use of the 'Collapse' rule on line 15.

Proof of [D6]:

1. [Variables] $(y_1, y_2, y_3, x_1, x_2, x_3)$
2. [Assumption] $(< (y_1, y_2, y_3, n_1, n_2) = func(x_1, x_2, x_3, n_1, n_2) >$
3. [Mf]
4. [$\forall$ Elim 3] $func(x_1, x_2, x_3, n_1, n_2) >= Wfunc1(Afunc4(Afunc5(Afunc6(x_1, x_2, x_3, n_1, n_2))), w)$
5. [= Sub 2, 4] $(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(Afunc4(Afunc5(Afunc6(x_1, x_2, x_3, n_1, n_2))), w) >$
6. [Mf14]
7. [$\forall$ Elim, 6] $< Afunc6(x_1, x_2, x_3, n_1, n_2) = ((0), x_2, x_3, n_1, n_2) >$
8. [= Sub 5, 7] $< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(Afunc4(Afunc5((0), v_2, v_3, n_1, n_2), w)) >$
9. [Mf12]
10. [$\forall$ Elim, 9] $< Afunc5((0), x_2, x_3, n_1, n_2) = ((0), (1), x_3, n_1, n_2) >$
11. [= Sub 8, 10] $< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(Afunc4((0), (1), x_3, n_1, n_2), w) >$
12. [Mf10]
13. [$\forall$ Elim, 12] $< Afunc4((0), (1), x_3, n_1, n_2) = ((0), (1), (1), n_1, n_2) >$
14. [= Sub 11, 13] $< (y_1, y_2, y_3, n_1, n_2) = Wfunc1((0, 1, 1, n_1, n_2), w) >$
15. [collapse 14] $< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(0, 1, 1, n_1, n_2, w) >$
16. [Number Theory]$< 1 = fac(0) >$
17. [Number Theory]$< 1 = exp(n_1, 0) >$
18. [$\wedge$ Intro 16, 17] $(< 1 = fac(0) >) \wedge (< 1 = exp(n_1, 0) >)$
19. [$\wedge$ Intro 15, 18] $(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(0, 1, 1, n_1, n_2, w) >) \wedge (< 1 = fac(0) >) \wedge (< 1 = exp(n_1, 0) >)$
20. [D5 Rename] : $\forall(w, v_1, v_2, v_3, w_1, w_2, w_3)$: $(< (v_1, v_2, v_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, w) > \wedge (< w_3 = fac(w_1) > \wedge < w_2 = exp(n, w_1) >)) \longrightarrow (< v_3 = fac(v_1) > \wedge < v_2 = exp(n, v_1) >)$
21. [$\forall$ Elim 20] : $(< (y_1, y_2, y_3, n_1, n_2) = Wfunc1(0, 1, 1, n_1, n_2, w) > \wedge (< 1 = fac(0) > \wedge < 1 = exp(n, 0) >)) \longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
22. [$\longrightarrow$ Elim 19, 21] $(< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
23. [$\longrightarrow$ Intro 2, 3-22] $(< (y_1, y_2, y_3, n_1, n_2) = func(x_1, x_2, x_3, n_1, n_2) > \longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$
24. [$\forall$ Intro 1, 2-23] $\forall(y_1, y_2, y_3, x_1, x_2, x_3)$: $(< (y_1, y_2, y_3, n_1, n_2) = func(x_1, x_2, x_3, n_1, n_2) > \longrightarrow (< y_3 = fac(y_1) > \wedge < y_2 = exp(n, y_1) >)$

6.4 The ACL2 Experiments

ACL2 is an automated reasoning system. It was developed at Computational Logic, Inc from 1988 to 1997, and development has continued at the University of Texas at Austin from then on. The ACL2 logic is a first-order logic of total recursive functions. It also supports mathematical induction on **N**. As part of this thesis, we tested some modified SMP outputs (ACL2 logic instead of S_L) on the ACL2 system. In this section, we present the results of these experiments.

6.4.1 ACL2 Vocabulary and Background

ACL2 can be used in two different modes ':program' and ':logic'. For our purposes, we are only interested in the :logic mode. In the :logic mode, any new function that is successfully defined is also axiomatically defined in the ACL2 logic. In order to define a function in the :logic mode, the 'defun' form is used. Proposed function definitions then go through syntactic and semantic checking. If the function is recursive, then ACL2 needs to find a proof that the function terminates in order to accept it. This means that it must find some decreasing 'measure' in the function's arguments. ACL2 understands the decreasing in the measure through the definition of a 'well-founded' ordering. ACL2 uses lisp syntax.

Example 6.4.1 This is the classical example of a function that appends two lists. At the command prompt, the user enters :

```
ACL2 !>(defun my-app(x y)
        (if
              (atom x) y
              (cons (car x)(my-app (cdr x) y))))
```

Remark 6.4.2 *atom* takes an argument and returns false if that argument is a list (or a pair). *cons* creates a pair from two arguments. *car* returns the first element of a list, *cdr* returns the tail end of a list. A list is a pair in which the second element is a list (which is empty at the end of it).

ACL2 responds with :

```
The admission of MY-APP is trivial, using the relation EO-ORD-< (which
is known to be well-founded on the domain recognized by EO-ORDINALP)
and the measure (ACL2-COUNT X).  We observe that the type of MY-APP
is described by the theorem (OR (CONSP (MY-APP X Y)) (EQUAL (MY-APP X Y) Y)).
We used primitive type reasoning.

Summary
Form:  ( DEFUN MY-APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.06 seconds (prove: 0.00, print: 0.02, other: 0.04)
 MY-APP
```

Indicating that it recognizes the 'cdr' function call passed as argument to the 'my-app' recursive call as a diminishing measure of in the argument list.

Accepted functions are then stored in the ACL2 database. From then on. they can be evaluated on arguments. or they can be used within proof contexts. Defining a function is one of a series of possible *events*. Another event that is of particular interest to us is of course the proving of a property of a function. To prove properties, we use the 'defthm' form.

Example 6.4.3 For example, to prove that the added length of two lists is equal to the length of the list that is the result of the 'my-app' function. we enter :

```
ACL2 !> (defthm my-app-length
            (equal (len (my-app x y))(+ (len x)(len y))))
```

To which ACL2 replies :

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
Suggested by this conjecture.  These merge into two derived induction
Schemes.  However, one of these is flawed and so we are left with one
```

viable candidate.

We will induct according to a scheme suggested by (LEN X), but modified to accommodate (MY-APP X Y). If we let (:P X Y) denote *1 above then the induction scheme we'll use is

```
(AND (IMPLIES (NOT (CONSP X)) (:P X Y))
     (IMPLIES (AND (CONSP X) (:P (CDR X) Y))
              (:P X Y))).
```

This induction is justified by the same argument used to admit LEN, namely, the measure (ACL2-COUNT X) is decreasing according to the relation E0-ORD-< (which is known to be well-founded on the domain recognized by E0-ORDINALP). When applied to the goal at hand the above induction scheme produces the following two nontautological subgoals.

Subgoal *1/2

```
(IMPLIES (NOT (CONSP X))
         (EQUAL (LEN (MY-APP X Y))
                (+ (LEN X) (LEN Y)))).
```

But simplification reduces this to T, using the :definitions FIX, LEN and MY-APP, primitive type reasoning, the :rewrite rule UNICITY-OF-0 and the :type-prescription rule LEN.

Subgoal *1/1

```
(IMPLIES (AND (CONSP X)
              (EQUAL (LEN (MY-APP (CDR X) Y))
                     (+ (LEN (CDR X)) (LEN Y))))
         (EQUAL (LEN (MY-APP X Y))
                (+ (LEN X) (LEN Y)))).
```

This simplifies, using the :definitions LEN and MY-APP, primitive type reasoning and the :rewrite rules CDR-CONS and COMMUTATIVITY-OF-+, to

Subgoal *1/1'

```
(IMPLIES (AND (CONSP X)
              (EQUAL (LEN (MY-APP (CDR X) Y))
                     (+ (LEN Y) (LEN (CDR X)))))
         (EQUAL (+ 1 (LEN (MY-APP (CDR X) Y)))
                (+ (LEN Y) 1 (LEN (CDR X))))).

But simplification reduces this to T, using linear arithmetic, primitive
type reasoning and the :type-prescription rule LEN.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM MY-APP-LENGTH ...)
Rules: ((:DEFINITION FIX)
        (:DEFINITION LEN)
        (:DEFINITION MY-APP)
        (:FAKE-RUNE-FOR-LINEAR NIL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CDR-CONS)
        (:REWRITE COMMUTATIVITY-OF-+)
        (:REWRITE UNICITY-OF-0)
        (:TYPE-PRESCRIPTION LEN))
Warnings:  None
Time:  0.30 seconds (prove: 0.08, print: 0.18, other: 0.04)
 MY-APP-LENGTH
```

ACL2 usually produces rather lengthy and cryptic proofs, so from now on, we will only include the parts that we consider essential. Also, ACL2 is a very large and complex system, with a very voluminous documentation which we couldn't seriously consider describing in this thesis. The ACL2 commands, background theory and logic are described in details in ([13],[14],[15],[16]).

6.4.2 Experiments

To give the reader an idea of the kind of experiments we conducted. we will give a detailed account of a typical ACL2 session. We start with the following program :

```
;Raw code ====>
;=====================================================
;int Fac(int n)
;{
; int  res  = 1;
; int  ctr = 0;
;
; while(ctr < n)
; {
; ctr = ctr + 1;
; res = res * ctr;
; }
;
; return res;
;}
;=====================================================
```

Which we then normalize using the Nivea 'NormalizeProgram' command.

```
;Variable Normalization :
;res ==> x_1
;ctr ==> x_2
;return ==> x_3


;Normalization of <int Fac(int n)> ====>
;=====================================================
;Vars :x_1 x_2 x_3
;Return Var :x_3
;Constants :n_1
```

```
;Fac(n_1)
;{
;[5]x_1  = 1;
;[4]x_2 = 0;
;while(x_2 < n_1)
;{
;[3]x_2 = x_2 + 1;
;[2]x_1 = x_1 * x_2;
;}
;[1]x_3 = x_1;
;}
;=================================================
```

Here, we use a modified version of the 'Modelize' command to create Code-Equivalent functions of the program. The version that we use creates a function that handles loop completion. We didn't include the theory for this into the thesis because it is only applicable to some programs (single top level while loop). This version also create 'groupings' of same level functions that it then considers as functions themselves. Finally, this version creates a list of function names, doesn't create $\exists$ expressions. and the output is in text format instead of Latex. The basic algorithm (SMP) however is the same. The output is :

```
;Model Translation Algorithm Output ====>
; <x_1 = 1>          ==> <AFac5(x_1,x_2,x_3,n_1) = (1,x_2,x_3,n_1)>
; <x_2 = 0>          ==> <AFac4(x_1,x_2,x_3,n_1) = (x_1,0,x_3,n_1)>
; <x_2 = x_2 + 1>    ==> <AFac3(x_1,x_2,x_3,n_1) = (x_1,x_2 + 1,x_3,n_1)>
; <x_1 = x_1 * x_2> ==> <AFac2(x_1,x_2,x_3,n_1) = (x_1 * x_2,x_2,x_3,n_1)>
; <x_3 = x_1>        ==> <AFac1(x_1,x_2,x_3,n_1) = (x_1,x_2,x_1,n_1)>

; <------->
; x_1  = 1
; x_2 = 0
; <------->          ==> <GFac2(x_1,x_2,x_3,n_1) = AFac4(AFac5(x_1,x_2,x_3,n_1))>
```

```
; <------->
; {
; x_2 = x_2 + 1
; x_1 = x_1 * x_2
; }
; <------->          ==> <GFac1(x_1,x_2,x_3,n_1) = AFac2(AFac3(x_1,x_2,x_3,n_1))>



;Modelization of <Fac(n_1)> ====>
;=================================================================
;=================================================================
;Variables :x_1 x_2 x_3
;Return Var :x_3
;Constants : n_1 1
;Function Variables: AFac1 AFac2 AFac3 AFac4 AFac5 GFac1 GFac2 WFac1a WFac1b

;Model Axioms:
; AFac1(x_1,x_2,x_3,n_1) = (x_1,x_2,x_1,n_1)
; AFac2(x_1,x_2,x_3,n_1) = (x_1 * x_2,x_2,x_3,n_1)
; AFac3(x_1,x_2,x_3,n_1) = (x_1,x_2 + 1,x_3,n_1)
; AFac4(x_1,x_2,x_3,n_1) = (x_1,0,x_3,n_1)
; AFac5(x_1,x_2,x_3,n_1) = (1,x_2,x_3,n_1)
; GFac1(x_1,x_2,x_3,n_1) = AFac2(AFac3(x_1,x_2,x_3,n_1))
; GFac2(x_1,x_2,x_3,n_1) = AFac4(AFac5(x_1,x_2,x_3,n_1))


;While Function :
; (x_4 = 0) ==> (WFac1a(x_1, x_2, x_3,n_1,x_4) =
;                        (x_1, x_2, x_3,n_1))
; (0 < x_4) ==> (WFac1a(x_1, x_2, x_3,n_1,x_4) =
;                        GFac1(WFac1a(x_1, x_2, x_3,n_1,x_4 - 1)))
; NOT(x_2 < n_1) ==> (WFac1b(x_1, x_2, x_3, n_1) = 0)
```

```
; (x_2 < n_1) ==> (WFac1b(x_1, x_2, x_3, n_1) =
;                     (1 + WFac1b(GFac2(x_1, x_2, x_3, n_1 - 1))))

;1 = WFac1b(GFac3(0, 0, 0, n_1))

;Fac(x_1, x_2, x_3, n_1) = GFac1(WFac1((GFac3(0, 0, 0, n_1)), 1))

;=====================================================================
;=====================================================================
```

The next step is to translate the definitions into the ACL2 language (lisp). This can be done automatically via a program. This causes no problem for functions built on assignments because they are not recursive. For example. $AFac1$ is translated as :

```
(defun AFac1 (arguments)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(list x_1 x_2 x_1 n_1)

))
```

This is a function that takes a list called 'arguments' as argument, and returns a list that is the application of the $AFac1$ function as defined in the Model to the elements of the list called arguments.

Remark 6.4.4 There are many different ways to do the translation. We chose this one because it is easy to automate. More involved translations also define the types of the arguments as integers.

ACL2 replies the following to the function definition :

```
Since AFAC1 is non-recursive, its admission is trivial.  We observe
that the type of AFAC1 is described by the theorem
(AND (CONSP (AFAC1 ARGUMENTS)) (TRUE-LISTP (AFAC1 ARGUMENTS))).  We
used primitive type reasoning.

Summary
Form:  ( DEFUN AFAC1 ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.06 seconds (prove: 0.00, print: 0.01, other: 0.05)
 AFAC1
```

Similarly, we translate and define the rest of the base case functions :

```
(defun AFac2 (arguments)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(list (* x_1 x_2) x_2 x_3 n_1)

))


(defun AFac3 (arguments)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
```

```
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(list x_1  (+ x_2 1) x_3 n_1)

))


(defun AFac4 (arguments)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(list x_1 0 x_3 n_1)

))


(defun AFac5 (arguments)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(list 1 x_2 x_3 n_1)
```

```
))
```

The second step is to define the group functions (composition of base case functions). For example, $GFac1$ is defined as :

```
(defun GFac1 (arguments)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(AFac2(AFac3(list x_1 x_2 x_3 n_1)))

))
```

To which ACL2 replies :

```
Since GFAC1 is non-recursive, its admission is trivial.  We observe
that the type of GFAC1 is described by the theorem
(AND (CONSP (GFAC1 ARGUMENTS)) (TRUE-LISTP (GFAC1 ARGUMENTS))).  We
used the :type-prescription rule AFAC2.

Summary
Form:  ( DEFUN GFAC1 ...)
Rules: ((:TYPE-PRESCRIPTION AFAC2))
Warnings:  None
Time:  0.07 seconds (prove: 0.00, print: 0.00, other: 0.07)
 GFAC1
```

Similarly we define $GFac2$:

```
(defun GFac2 (arguments)
(let
```

```
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(AFac4(AFac5(list x_1 x_2 x_3 n_1)))

))
```

We are now ready to define the first part of the loop encoding. We do this like so :

```
(defun WFac1a (arguments w)
(let
(
(x_1 (mv-nth 0 arguments))
(x_2 (mv-nth 1 arguments))
(x_3 (mv-nth 2 arguments))
(n_1 (mv-nth 3 arguments))
)

(if (zp w)
  (list x_1 x_2 x_3 n_1)
(GFac1(WFac1a (list x_1 x_2 x_3 n_1) (- w 1)))
)


))
```

We added another argument to describe the extra variable 'w'.

Remark 6.4.5 'zp' is the 'IfZero' function.

ACL2 replies :

```
The admission of WFAC1A is trivial, using the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT W).  We observe that the type of WFAC1A
is described by the theorem
(AND (CONSP (WFAC1A ARGUMENTS W)) (TRUE-LISTP (WFAC1A ARGUMENTS W))).
We used primitive type reasoning and the :type-prescription rule GFAC1.

Summary
Form:  ( DEFUN WFAC1A ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:TYPE-PRESCRIPTION GFAC1))
Warnings:  None
Time:  0.07 seconds (prove: 0.00, print: 0.02, other: 0.05)
 WFAC1A
ACL2 !>>Bye.
 :EOF
```

In order to prove correctness properties on our function definition, we need to define the factorial function. We use the classical definition :

```
(defun fac (n)
(if (zp n) 1 (* n (fac(- n 1))))

)
```

With resulting ACL2 output :

```
The admission of FAC is trivial, using the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT N).  We observe that the type of FAC is
described by the theorem (AND (INTEGERP (FAC N)) (< 0 (FAC N))).  We
used the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and primitive
type reasoning.

Summary
Form:  ( DEFUN FAC ...)
```

```
Rules: ((:COMPOUND-RECOGNIZER ZP-COMPOUND-RECOGNIZER)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.07 seconds (prove: 0.00, print: 0.04, other: 0.03)
 FAC
```

We start by proving the equivalent of the S_L expression:

$\forall(y_1, y_2, y_3, x_1, x_2, x_3) :< (y_1, y_2, y_3) = GFac2(x_1, x_2, x_3) > \longrightarrow (< x_1 = fac(x_2) >)$

This is written in ACL2 like so :

```
(defthm correc1

(IMPLIES (AND

(EQUAL (mv-nth 0 (GFac2 Tuple)) x_1)
(EQUAL (mv-nth 1 (GFac2 Tuple)) x_2)

)

(EQUAL (fac x_2) x_1)))
```

This translates as : For any tuple, if x_1 is the first element of the output list of $GFac2$ and x_2 is the second one, then $x_1 = fac(x_2)$. This makes ACL2 produce the following proof :

```
ACL2 Warning [Free] in ( DEFTHM CORREC1 ...):  The :REWRITE rule generated
from CORREC1 contains the free variables X_1 and TUPLE.  These variables
will be chosen by searching for an instance of
(EQUAL (MV-NTH 0 (GFAC2 TUPLE)) X_1) among the hypotheses of the conjecture
being rewritten.  This is generally a severe restriction on the applicability
of the :REWRITE rule.


ACL2 Warning [Non-rec] in ( DEFTHM CORREC1 ...):  As noted, we will
```

```
instantiate the free variables, X_1 and TUPLE, of the :REWRITE rule
CORREC1, by searching for the hypothesis shown above.  However, this
hypothesis mentions the function symbol GFAC2, which is defun'd non-
recursively. Unless disabled, this function symbol is unlikely to occur
in the conjecture being proved and hence the search for the required
hypothesis will likely fail.


But simplification reduces this to T, using the :definitions AFAC4,
AFAC5, GFAC2 and MV-NTH, the :executable-counterparts of BINARY-+,
EQUAL, FAC and ZP, primitive type reasoning and the :rewrite rules
CAR-CONS and CDR-CONS.

Q.E.D.

Summary
Form:  ( DEFTHM CORREC1 ...)
Rules: ((:DEFINITION AFAC4)
        (:DEFINITION AFAC5)
        (:DEFINITION GFAC2)
        (:DEFINITION MV-NTH)
        (:EXECUTABLE-COUNTERPART BINARY-+)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART FAC)
        (:EXECUTABLE-COUNTERPART ZP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS))
Warnings:  Non-rec and Free
Time:  0.19 seconds (prove: 0.05, print: 0.03, other: 0.11)
 CORREC1
```

We now show some key parts of the ACL2 equivalent of the general induction proof on the while loop (This proof on the factorial function while encoding was covered in Chapter 4, proof 6). The ACL2 expression is :

```
(defthm correc2

(IMPLIES (AND

(EQUAL (mv-nth 0 Tuple) x_1)
(EQUAL (mv-nth 1 Tuple) x_2)
(EQUAL (fac x_2) x_1)
(EQUAL (mv-nth 0 (GFac1 Tuple)) y_1)
(EQUAL (mv-nth 1 (GFac1 Tuple)) y_2)
(integerp x_1)
(integerp x_2)
(integerp y_1)
(integerp y_2)
(integer-listp Tuple)
(integer-listp (GFac1 Tuple))

)

(EQUAL (fac y_2) y_1))

:hints
(("Subgoal *1.2/1.1" : use fac-axiom1))
:in-theory (enable fac-axiom2)))
:otf-flg t)
```

'fac-axiom1' and 'fac-axiom2' are the equivalent of the non-proven [Number Theory] axioms that we use in the hand proofs. One is for numbers, the other is for lists. These are given as hints to the prover for help in finding a proof. We also set the :otf-flg flag to true (onward through the fog), forcing ACL to go back and explore all sub cases.
Some key parts of the output ACL2 proof are presented in Appendix B.

6.4.3 General Discussion of the Results of the ACL2 Experiments

In general, we were successful on proving important parts of the property proofs that were done by hand using ACL2 when working on simple programs (exponential, fibonnaci,...). However, in a lot of cases, important hints needed to be provided to the theorem provers, and an in depth analysis of the tried and failed proof trees needed to be conducted. The completeness (loop ending) attempts all failed. The by default induction schemes that are provided with ACL2 were not adapted (well-founded orderings inadequate). Perhaps there is a general well-ordered-relation that might work, but we have not discovered it. However, we achieved a good ratio of success/trials (3 successes on 5 tries) when dealing with programs that contained only 1 top level while loop. The programs that we were successful in proving correctness properties of are the factorial program, the exponential program, and the Chapter 1 example program. While we did manage to prove some properties of the fibonnaci program and of a fibonnaci variant program, we were unsuccessful in proving the key correctness properties for these (proof that the desired computation is completed). Proofs on programs that contained no while loops were all successful.

Chapter 7

Conclusions

7.1 Summary of Contributions

In this thesis we addressed the problem of finding an algorithmic way to translate programs in a C like language into Models in a fully deductive logical system. This is a valid problem to address because a solution to it allows the deriving of formal proofs of properties on the Models which can then in turn easily be translated into proofs of properties of the programs themselves. We solved this problem by building a logical system (S_L) that was adapted to our purpose, and by constructing an algorithm which takes programs as input and outputs Models (SMP). The system we presented allows the derivation of proofs of properties on code segments independently of the segments themselves. While the Models that are constructed by the SMP algorithm do not allow us to derive proofs of all the properties of the programs, they do allow the derivations of the properties that do not depend on loop termination. We gave a formal justification as to why derivations constructed on SMP built Models are sound. We also showed how proofs can be assembled together using a lemma approach. Finally, for the experimental side of our work, we implemented the SMP algorithm and we demonstrated that proofs could be produced by theorem provers that were not designed for the purpose of program correctness. We used the theorem prover ACL2 as an experimental tool to illustrate this. We also presented a complete example of our method (the $func$ analysis).

The major contributions of the thesis are:

1. The S_L logic. A logic system that permits the definition of recursive functions while allowing formal proofs of the properties of those functions.

2. The SMP algorithm. An algorithm which outputs a Model in S_L logic using a program as input, where a Model is a set of Tuple-Property Class definitions that are Code-Equivalent to the statements of the program used as input.

The lesser contributions of the thesis are:

1. An original proof that a 'while' programming language (such as C^c) generates recursive functions (while there are other proofs of this, this one demonstrates the use of modern formal methods as a mathematical tool capable of proofs on the theory of computer science). This proof can be found in Chapter 3.

2. A set of definitions for Code-Equivalent mathematical functions, and 2 associated lemmas that demonstrate the usefulness of the concept in the context of program correctness proofs (chapter 5).

3. An implementation of the SMP algorithm.

What the thesis does not cover:

Encodings that would allow the derivation of proofs of properties that stems from the termination or from the conditional expression of a loop in a code segment. We didn't tackle completeness.

7.2 Comparisons with Existing Work

7.2.1 Verification of Flow Chart Programs

Verification by *flowcharts* is a proof system that deals with a class of program that is similar complexity wise to C^c. The system is described in ([18].

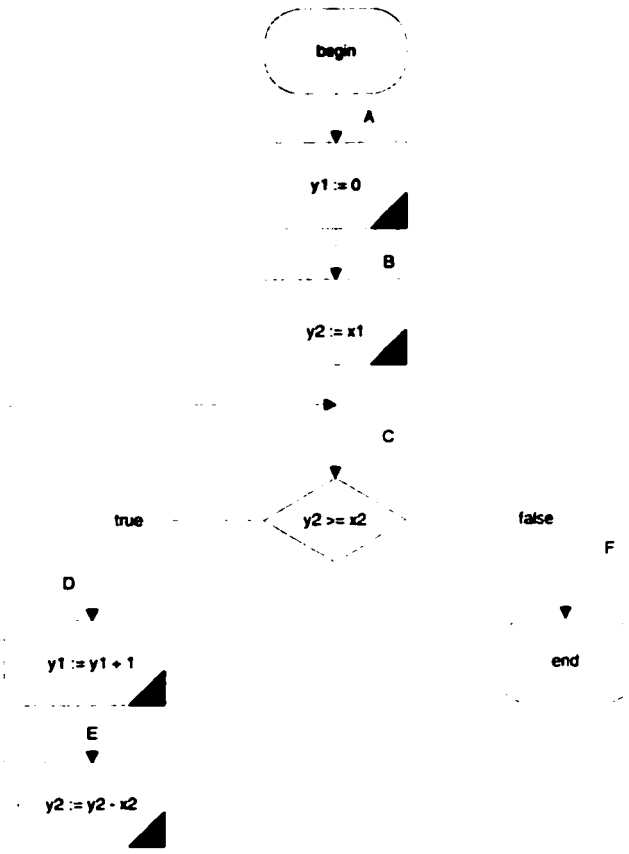

Figure 7.1: Integer Division Flowchart

Chapter 7 pg180-pg195) and in ([11], Chapter1, pg 61-pg 65). Programs are built from simple assignments, of the form $v := e$ where v is a program variable and e is a first order term. There are also decision predicates, which are unquantified first order formulas over the whole structure. The whole structure is then represented visually. We reproduce the structure that is given as an example in figure 7.2 of ([18]) in Figure 7.1. This flowchart represents an integer division program. Invariants (expressions in some logic system) are then assigned to each labeled transitions (A to F). In this case, a suitable choice (as given in ([18], pg186)) would be:

$\varphi(A) = x_1 \geq 0 \wedge x_2 > 0$

$\varphi(B) = x_1 \geq 0 \wedge x_2 > 0 \wedge y_1 \equiv 0$
$\varphi(C) = (x_1 \equiv y_1 * x_2 + y_2) \wedge y_2 \geq 0$
$\varphi(D) = (x_1 \equiv y_1 * x_2 + y_2) \wedge y_2 \geq x_2$
$\varphi(E) = (x_1 \equiv y_1 * x_2 + y_2 - x_2) \wedge y_2 - x_2 \geq 0$
$\varphi(F) = (x_1 \equiv y_1 * x_2 + y_2) - x_2) \wedge y_2 \geq 0 \wedge y_2 < x_2$

The text doesn't give a formal explanation as to how these invariants are assigned, but once they are assigned, a consistency proof can be given by showing the following implications (where c is the flowchart arrow. $pre(c)$ is the predicate that holds before c and $post(c)$ is the predicate that holds after c):

- If c is a positive decision (such as the D arrow in the diagram).

  $$(pre(c) \wedge p) \longrightarrow post(c)$$

  needs to be proven (where p is the decision predicate).

- If c is a negative decision,

  $$(pre(c) \wedge \neg p) \longrightarrow post(c)$$

  needs to be proven (F arrow).

- If c is an assignment,

  $$pre(c) \longrightarrow post(c)[e/v]$$

  needs to be proven (A, B and C arrows).

In the example's case, the proof is produced by proving the expressions below:

$\varphi(A) \longrightarrow \varphi(B)[0/y_1]$
$\varphi(B) \longrightarrow \varphi(C)[x_1/y_2]$
$(\varphi(C) \wedge y_2 \geq x_2) \longrightarrow \varphi(D)$
$(\varphi(C) \wedge \neg y_2 \geq x_2) \longrightarrow \varphi(F)$
$\varphi(D) \longrightarrow \varphi(E)[y_1 + 1/y_1]$
$\varphi(E) \longrightarrow \varphi(C)[y_2 - x_2/y_2]$

Advantages of this Method over our Approach :

This method is simpler than ours. The programs are represented graphically, which increases the general understanding of what the computation they perform is, and how it is performed. It also gives a nice representation of the state changes that occur during a program's execution.

Disadvantages of this Method over our Approach :

- Large programs would be difficult to represent using this method. There is no real formal way to generate the invariants as these are separate goals whose proof needs to be produced from one goal to the next. While it is a good method to use in conjunction with the actual writing of the program, the format is difficult to formalize.

- The proofs cannot be separated from the charts. This makes automatic proving difficult. A data structure that represent the flowchart would need to be implemented (then again, it could be argued that a program is already itself such a data structure).

- Because it is a state based approach, all the information about the current state that is being analyzed is coded into the expression that represents it. However, no general information about the functions themselves is available into the relevant expression. In other word, only transition information is encoded in the proof expressions.

7.2.2 Hoare Triple Logic

We have already used and described Hoare Triple Logic extensively throughout the thesis, so in this section, we will only do a comparison between it and our method.

Advantages of H.T.L. over our method

- The Hoare Triple style of correctness proving is shorter, simpler and clearer.

- It is easier to directly relate the code to the proof when using H.T.L. because the format alternates between the program statements and the logical expressions.

- Hoare Triple Logic is easier to use in conjunction with the programming itself. Creating the proof at the same time the program is being written is a simpler matter than with our method because of the second point of this comparison.

Disadvantages of H.T.L. over our method

- In our method, properties can be proved separately from the program. Once the Model has been created, there is a variety of methods that can be used to prove a specific property.

- It is easier to reuse some of the work that is done when using our method, because proofs can be broken into lemmas. Proof components can then be reassembled together. As an example of this, consider proof 6 of the *func* analysis in Chapter 6. A substantial part of the proof could be eliminated by proving that :

  $\forall(y_1, y_2, y_3, v_1, v_2, v_3) :< (y_1, y_2, y_3, n_1, n_2) =$
  $Afunc4(Afunc5(Afunc6(v_1, v_2, v_3, n_1, n_2))) >$
  $\longrightarrow< (y_1, y_2, y_3, n_1, n_2) = (0, 1, 1, n_1, n_2) >$

  A simple proof of this is given below, and we notice that once this proof is built, it can simply be referred to and reused in any property proof in the context of *func*.

Proof 1:

1.[Vars] $(y_1, y_2, y_3, v_1, v_2, v_3)$
2. [Assumption] $< (y_1, y_2, y_3, n_1, n_2) = Afunc4(Afunc5(Afunc6(v_1, v_2, v_3, n_1, n_2))) >$
3. [Mf14]
4. [$\forall$ Elim, 3] $< Afunc6(v_1, v_2, v_3, n_1, n_2) = ((0), v_2, v_3, n_1, n_2) >$
5. [= Sub 2, 4] $< (y_1, y_2, y_3, n_1, n_2) = Afunc4(Afunc5((0), v_2, v_3, n_1, n_2)) >$
6. [Mf12]
7. [$\forall$ Elim, 6] $< Afunc5((0), v_2, v_3, n_1, n_2) = ((0), (1), v_3, n_1, n_2) >$
8. [= Sub 5, 7] $< (y_1, y_2, y_3, n_1, n_2) = Afunc4((0), (1), v_3, n_1, n_2) >$
9. [Mf10]
10. [$\forall$ Elim, 9] $< Afunc4((0), (1), v_3, n_1, n_2) = ((0), (1), (1), n_1, n_2) >$
11. [= Sub 8, 10] $< (y_1, y_2, y_3, n_1, n_2) = ((0), (1), (1), n_1, n_2) >$
12. [$\longrightarrow$ Intro 2,3-11] $< (y_1, y_2, y_3, n_1, n_2) = Afunc4(Afunc5(Afunc6(v_1, v_2, v_3, n_1, n_2))) >$
$\longrightarrow < (y_1, y_2, y_3, n_1, n_2) = (0, 1, 1, n_1, n_2) >$
13.[$\forall$ Intro 1,2-12]$\forall (y_1, y_2, y_3, v_1, v_2, v_3)$:
$< (y_1, y_2, y_3, n_1, n_2) = Afunc4(Afunc5(Afunc6(v_1, v_2, v_3, n_1, n_2))) >$
$\longrightarrow < (y_1, y_2, y_3, n) = (0, 1, 1, n_1, n_2) >$

- This also means that if the program is changed, it is easier to keep and to reuse proofs that are not affected by the changes.
- The Models are generated algorithmically, which means that Models of large programs are just as easy to generate as Models of small programs.
- The method can be modified to work in conjunction with most theorem proving tools that already exist.
- Because of these points, we think that our method has the potential to be easier to use than Hoare Triple Logic in the context of larger programs.

7.3 Future Work

7.3.1 Completeness

We have already begun to explore encodings that would allow the derivation of proofs of the property that are the result of loop completions. One of the encodings that we are considering is the one that given a while loop in the form :

```
while(Cond)
{
[WHILE BODY]
}
```

creates a function in the form :

- $Wf2(v_1, \ldots, v_p) = 0$ if $\neg(Cond)$

- $Wf2(v_1, \ldots, v_p) = Wf2(F_{Bod}(v_1, \ldots, v_p)) + 1$ otherwise
  (where F_{Bod} is the SMP encoding of [WHILE BODY])

This would formally translate in S_L to

- $\neg(F_{Cond}(v_1, \ldots, v_p)) \longrightarrow Wf2(v_1, \ldots, v_p) = 0$
- $(F_{Cond}(v_1, \ldots, v_p)) \longrightarrow Wf2(v_1, \ldots, v_p) = Wf2(F_{Bod}(x_1, \ldots, x_2)) + 1$

Where $F_{Cond} : \mathbb{N}^p \Longrightarrow \mathbb{B}$ would be an encoding for a function with boolean output. Completeness might then be formalized by proving :

$$\forall(x_1, \ldots, x_p) : \exists w : Wf2(x_1, \ldots, x_p) = w$$

7.3.2 C^c Language Extensions

Extending the C^c language to make it handle logical operators, the comparison operators($>, \geq, \leq$) and other data types (Booleans, arrays) would also be a useful addition to the work that we have done so far. There are encodings that we are considering for Booleans (either as primitives or as constant functions) and for arrays.

Subroutine (function call)

Extending both the language and the logical system to make it handle function calls would also work well with the philosophy of the work (reuse and modularity). This would also permit property derivations on recursive Code Segments.

Bibliography

[1] R. Dedekind. Was sind und was sollen die Zahlen?. Braunschweig. 1888

[2] S. K. Kleene. General recursive functions of natural numbers. Math. Ann 112, 1936 727-742

[3] R. Cori, D. Lascar. Logique Mathematique. Cours et exercises. Vol II. Masson, 1993

[4] A. Church, An unsolvable problem of elementary number theory. American Journal of Mathematics, 1936

[5] A. M. Turing, On computable numbers with an application to the Entscheidungsproblem, Proceedings of the London Mathematics Society. 1936

[6] C. C. Chang, H. Jerome Keisler, Model Theory. North-Holland. 1973

[7] R. R. Stoll. Set Theory And Logic. Dover. 1961

[8] CAR Hoare, An axiomatic basis for computer programming, Comm of the ACM, October 1969

[9] D. Hofstadter, Godel, Escher, Bach : An eternal Golden Braid. Basic Books, 1979

[10] K.R. Apt, Ten years of Hoare's logic : A survey, Transactions on Programming Languages and systems, 1981

[11] P. Odifreddi, Classical Recursion Theory, North-Holland. 1989

[12] C. H. Papadimitriou, Computational Complexity. Addison-Wesley. 1994.

[13] M. Kauffman and J. S. Moore, A precise Description of the ACL2 Logic, University of Texas at Austin, 1998

[14] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. Computer-Aided Reasoning: An Approach, Kluwer Academic Publishers. 2000

[15] Matt Kaufmann, Panagiotis Manolios, Computer-Aided Reasoning: ACL2 Case Studies, and J Strother Moore (eds.), Kluwer Academic Publishers, 2000

[16] Matt Kaufmann and J Moore,An Industrial Strength Theorem Prover for a Logic Based on Common Lisp, IEEE Transactions on Software Engineering 23(4), 1997

[17] M. R. A. Huth and M. D. Ryan, Logic in Computer Science : Modelling and Reasonning about Systems. Cambridge University Press. 2000

[18] D. A. Peled, Software Reliability Methods. Springer. 2001

Appendix A

ACL2 output for the main proof of Chapter 6

```
ACL2 Warning [Non-rec] in ( DEFTHM CORREC2 ...):  As noted, we will
instantiate the free variables, X_1, X_2, Y_1 and TUPLE, of the :REWRITE
rule CORREC2, by searching for the set of hypotheses shown above.
However, these hypotheses mention the function symbol GFAC1, which
is defun'd non-recursively. Unless disabled, this function symbol is
unlikely to occur in the conjecture being proved and hence the search
for the required hypotheses will likely fail.


This simplifies, using the :definitions AFAC2, AFAC3, FIX, GFAC1, INTEGER-
LISTP and MV-NTH, the :executable-counterparts of BINARY-+, INTEGER-
LISTP and ZP, primitive type reasoning, the :forward-chaining rules
INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-
TO-TRUE-LISTP, the :rewrite rules CAR-CONS, CDR-CONS, COMMUTATIVITY-
OF-*, COMMUTATIVITY-OF-+, DISTRIBUTIVITY and UNICITY-OF-1 and the :type-
prescription rules FAC, INTEGER-LISTP and RATIONAL-LISTP, to

Goal'
(IMPLIES (AND (CONSP TUPLE)
              (EQUAL (FAC (MV-NTH 1 TUPLE))
```

```
                (CAR TUPLE))
           (INTEGERP (MV-NTH 1 TUPLE))
           (INTEGER-LISTP TUPLE)
           (INTEGERP (MV-NTH 2 TUPLE))
           (INTEGERP (MV-NTH 3 TUPLE)))
      (EQUAL (FAC (+ 1 (MV-NTH 1 TUPLE)))
             (+ (CAR TUPLE)
                (* (CAR TUPLE) (MV-NTH 1 TUPLE))))).
```

The destructor terms (CAR TUPLE) and (CDR TUPLE) can be eliminated by using CAR-CDR-ELIM to replace TUPLE by (CONS TUPLE1 TUPLE2), generalizing (CAR TUPLE) to TUPLE1 and (CDR TUPLE) to TUPLE2 and restricting the type of the new variable TUPLE1 to be that of the term it replaces, as established by the :type-prescription rule FAC. This produces the following goal.

[Note: A hint was supplied for our processing of the goal below. Thanks!]

Goal''

```
(IMPLIES (AND (INTEGERP TUPLE1)
              (< 0 TUPLE1)
              (CONSP (CONS TUPLE1 TUPLE2))
              (EQUAL (FAC (MV-NTH 1 (CONS TUPLE1 TUPLE2)))
                     TUPLE1)
              (INTEGERP (MV-NTH 1 (CONS TUPLE1 TUPLE2)))
              (INTEGER-LISTP (CONS TUPLE1 TUPLE2))
              (INTEGERP (MV-NTH 2 (CONS TUPLE1 TUPLE2)))
              (INTEGERP (MV-NTH 3 (CONS TUPLE1 TUPLE2))))
         (EQUAL (FAC (+ 1 (MV-NTH 1 (CONS TUPLE1 TUPLE2))))
                (+ TUPLE1
                   (* TUPLE1
                      (MV-NTH 1 (CONS TUPLE1 TUPLE2)))))).
```

This simplifies, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+, INTEGERP, MV-NTH and ZP, primitive type reasoning, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP, the :rewrite rules CAR-CONS and CDR-CONS and the :type-prescription rules FAC, INTEGER-LISTP and RATIONAL-LISTP, to

```
Goal'''
(IMPLIES (AND (< 0 TUPLE1)
              (CONSP TUPLE2)
              (EQUAL (FAC (CAR TUPLE2)) TUPLE1)
              (INTEGERP (CAR TUPLE2))
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
              (INTEGERP (MV-NTH 2 TUPLE2)))
         (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                (+ TUPLE1 (* TUPLE1 (CAR TUPLE2))))).
```

This simplifies, using the :rewrite rule COMMUTATIVITY-OF-*, to

```
Goal'4'
(IMPLIES (AND (< 0 (FAC (CAR TUPLE2)))
              (CONSP TUPLE2)
              (INTEGERP (CAR TUPLE2))
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
              (INTEGERP (MV-NTH 2 TUPLE2)))
         (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                (+ (FAC (CAR TUPLE2))
                   (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).
```

Name the formula above *1.

Perhaps we can prove *1 by induction. One induction scheme is suggested

```
by this conjecture.

We will induct according to a scheme suggested by (INTEGER-LISTP TUPLE2).
If we let  (:P TUPLE2) denote *1 above then the induction scheme we'll
use is
(AND (IMPLIES (AND (NOT (ATOM TUPLE2))
                   (NOT (INTEGERP (CAR TUPLE2))))
              (:P TUPLE2))
     (IMPLIES (AND (NOT (ATOM TUPLE2))
                   (INTEGERP (CAR TUPLE2))
                   (:P (CDR TUPLE2)))
              (:P TUPLE2))
     (IMPLIES (ATOM TUPLE2) (:P TUPLE2))).
This induction is justified by the same argument used to admit INTEGER-
LISTP, namely, the measure (ACL2-COUNT TUPLE2) is decreasing according
to the relation E0-ORD-< (which is known to be well-founded on the
domain recognized by E0-ORDINALP).  When applied to the goal at hand
the above induction scheme produces the following seven nontautological
subgoals.

.
.
.

The destructor terms (CAR TUPLE2) and (CDR TUPLE2) can be eliminated.
Furthermore, those terms are at the root of a chain of two rounds of
destructor elimination. (1) Use CAR-CDR-ELIM to replace TUPLE2 by
(CONS TUPLE3 TUPLE4), generalizing (CAR TUPLE2) to TUPLE3 and (CDR TUPLE2)
to TUPLE4.  (2) Use CAR-CDR-ELIM, again, to replace TUPLE4 by
(CONS TUPLE5 TUPLE6), generalizing (CAR TUPLE4) to TUPLE5 and (CDR TUPLE4)
to TUPLE6.    These steps produce the following goal.

Subgoal *1/7'''
(IMPLIES (AND (CONSP (CONS TUPLE5 TUPLE6))
```

```
            (CONSP (LIST* TUPLE3 TUPLE5 TUPLE6))
            (INTEGERP TUPLE3)
            (EQUAL (FAC (+ 1 TUPLE5))
                   (+ (FAC TUPLE5)
                      (* TUPLE5 (FAC TUPLE5))))
            (< 0 (FAC TUPLE3))
            (INTEGER-LISTP (CONS TUPLE5 TUPLE6))
            (INTEGERP TUPLE5)
            (INTEGERP (MV-NTH 1 (CONS TUPLE5 TUPLE6))))
      (EQUAL (FAC (+ 1 TUPLE3))
             (+ (FAC TUPLE3)
                (* TUPLE3 (FAC TUPLE3))))).
```

This simplifies, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+ and ZP, primitive type reasoning, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP, the :rewrite rules CAR-CONS and CDR-CONS and the :type-prescription rules INTEGER-LISTP and RATIONAL-LISTP, to

.
.
.

We generalize this conjecture, replacing (FAC TUPLE3) by I and restricting the type of the new variable I to be that of the term it replaces, as established by FAC. This produces

```
Subgoal *1/7'5'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (EQUAL (FAC (+ 1 TUPLE5))
                     (+ (FAC TUPLE5)
```

```
                    (* TUPLE5 (FAC TUPLE5))))
             (INTEGER-LISTP TUPLE6)
             (INTEGERP TUPLE5)
             (CONSP TUPLE6)
             (INTEGERP (CAR TUPLE6)))
        (EQUAL (FAC (+ 1 TUPLE3))
               (+ I (* TUPLE3 I)))).
```

This simplifies, using the :rewrite rule COMMUTATIVITY-OF-*, to

```
Subgoal *1/7'6'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (EQUAL (FAC (+ 1 TUPLE5))
                     (+ (FAC TUPLE5)
                        (* TUPLE5 (FAC TUPLE5))))
              (INTEGER-LISTP TUPLE6)
              (INTEGERP TUPLE5)
              (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

Name the formula above *1.1.

```
Subgoal *1/6
(IMPLIES (AND (NOT (ATOM TUPLE2))
              (INTEGERP (CAR TUPLE2))
              (NOT (INTEGERP (MV-NTH 2 (CDR TUPLE2))))
              (< 0 (FAC (CAR TUPLE2)))
              (CONSP TUPLE2)
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
```

```
            (INTEGERP (MV-NTH 2 TUPLE2)))
          (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                 (+ (FAC (CAR TUPLE2))
                    (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).
```

By the simple :definition ATOM we reduce the conjecture to

```
Subgoal *1/6'
(IMPLIES (AND (CONSP TUPLE2)
              (INTEGERP (CAR TUPLE2))
              (NOT (INTEGERP (MV-NTH 2 (CDR TUPLE2))))
              (< 0 (FAC (CAR TUPLE2)))
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
              (INTEGERP (MV-NTH 2 TUPLE2)))
         (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                (+ (FAC (CAR TUPLE2))
                   (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).

.
.
.

Subgoal *1/6'''
(IMPLIES (AND (CONSP (CONS TUPLE5 TUPLE6))
              (CONSP (LIST* TUPLE3 TUPLE5 TUPLE6))
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 2 (CONS TUPLE5 TUPLE6))))
              (< 0 (FAC TUPLE3))
              (INTEGER-LISTP (CONS TUPLE5 TUPLE6))
              (INTEGERP TUPLE5)
              (INTEGERP (MV-NTH 1 (CONS TUPLE5 TUPLE6))))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ (FAC TUPLE3)
```

```
                    (* TUPLE3 (FAC TUPLE3))))).
```

This simplifies, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+ and ZP, primitive type reasoning, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP, the :rewrite rules CAR-CONS and CDR-CONS and the :type-prescription rules INTEGER-LISTP and RATIONAL-LISTP, to

```
Subgoal *1/6'4'
(IMPLIES (AND (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (< 0 (FAC TUPLE3))
              (INTEGER-LISTP TUPLE6)
              (INTEGERP TUPLE5)
              (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ (FAC TUPLE3)
                   (* TUPLE3 (FAC TUPLE3))))).
```

We generalize this conjecture, replacing (FAC TUPLE3) by I and restricting the type of the new variable I to be that of the term it replaces, as established by FAC. This produces

```
Subgoal *1/6'5'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (INTEGER-LISTP TUPLE6)
              (INTEGERP TUPLE5)
              (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6)))
```

```
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* TUPLE3 I)))).
```

This simplifies, using the :rewrite rule COMMUTATIVITY-OF-*, to

```
Subgoal *1/6'6'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (INTEGER-LISTP TUPLE6)
              (INTEGERP TUPLE5)
              (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

We suspect that the term (INTEGERP TUPLE5) is irrelevant to the truth of this conjecture and throw it out. We will thus try to prove

```
Subgoal *1/6'7'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (INTEGER-LISTP TUPLE6)
              (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

Name the formula above *1.2.

Subgoal *1/5

```
(IMPLIES (AND (NOT (ATOM TUPLE2))
              (INTEGERP (CAR TUPLE2))
              (NOT (INTEGERP (MV-NTH 1 (CDR TUPLE2))))
              (< 0 (FAC (CAR TUPLE2)))
              (CONSP TUPLE2)
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
              (INTEGERP (MV-NTH 2 TUPLE2)))
         (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                (+ (FAC (CAR TUPLE2))
                   (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).
```

By the simple :definition ATOM we reduce the conjecture to

.
.
.

By the simple :definition ATOM we reduce the conjecture to

Subgoal *1/4'

```
(IMPLIES (AND (CONSP TUPLE2)
              (INTEGERP (CAR TUPLE2))
              (NOT (INTEGER-LISTP (CDR TUPLE2)))
              (< 0 (FAC (CAR TUPLE2)))
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
              (INTEGERP (MV-NTH 2 TUPLE2)))
         (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                (+ (FAC (CAR TUPLE2))
                   (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).
```

But simplification reduces this to T, using the :definition INTEGER-LISTP.

```
Subgoal *1/3
(IMPLIES (AND (NOT (ATOM TUPLE2))
              (INTEGERP (CAR TUPLE2))
              (NOT (INTEGERP (CADR TUPLE2)))
              (< 0 (FAC (CAR TUPLE2)))
              (CONSP TUPLE2)
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
              (INTEGERP (MV-NTH 2 TUPLE2)))
         (EQUAL (FAC (+ 1 (CAR TUPLE2)))
                (+ (FAC (CAR TUPLE2))
                   (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).
```

By the simple :definition ATOM we reduce the conjecture to

.
.
.

But simplification reduces this to T, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+ and ZP, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP and the :type-prescription rules INTEGER-LISTP and RATIONAL-LISTP.

```
Subgoal *1/2
(IMPLIES (AND (NOT (ATOM TUPLE2))
              (INTEGERP (CAR TUPLE2))
              (NOT (CONSP (CDR TUPLE2)))
              (< 0 (FAC (CAR TUPLE2)))
              (CONSP TUPLE2)
              (INTEGER-LISTP TUPLE2)
              (INTEGERP (MV-NTH 1 TUPLE2))
```

```
            (INTEGERP (MV-NTH 2 TUPLE2)))
       (EQUAL (FAC (+ 1 (CAR TUPLE2)))
              (+ (FAC (CAR TUPLE2))
                 (* (CAR TUPLE2) (FAC (CAR TUPLE2)))))).
```

By the simple :definition ATOM we reduce the conjecture to

Subgoal *1/2'

.
.
.

But simplification reduces this to T, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+, INTEGERP, MV-NTH and ZP, primitive type reasoning, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP and the :type-prescription rules INTEGER-LISTP and RATIONAL-LISTP.

.
.
.

This induction is justified by the same argument used to admit INTEGER-LISTP, namely, the measure (ACL2-COUNT TUPLE6) is decreasing according to the relation E0-ORD-< (which is known to be well-founded on the domain recognized by E0-ORDINALP). When applied to the goal at hand the above induction scheme produces the following four nontautological subgoals.

Subgoal *1.2/4

```
(IMPLIES (AND (NOT (ATOM TUPLE6))
              (INTEGERP (CAR TUPLE6))
              (NOT (INTEGERP (CADR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
```

```
            (INTEGERP TUPLE3)
            (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
            (INTEGER-LISTP TUPLE6)
            (CONSP TUPLE6))
       (EQUAL (FAC (+ 1 TUPLE3))
              (+ I (* I TUPLE3)))).
```

By the simple :definition ATOM we reduce the conjecture to

.
.
.

This simplifies, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+ and ZP, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP and the :type-prescription rules INTEGER-LISTP and RATIONAL-LISTP, to

```
Subgoal *1.2/4''
(IMPLIES (AND (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6))
              (NOT (INTEGERP (CADR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (INTEGER-LISTP (CDR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

This simplifies, using the :definition INTEGER-LISTP and primitive type reasoning, to

.

.
.

This simplifies, using the :executable-counterparts of CAR, CONSP and INTEGERP, to

```
Subgoal *1.2/4'4'
```

.
.
.

The destructor terms (CAR TUPLE6) and (CDR TUPLE6) can be eliminated by using CAR-CDR-ELIM to replace TUPLE6 by (CONS TUPLE7 TUPLE8), generalizing (CAR TUPLE6) to TUPLE7 and (CDR TUPLE6) to TUPLE8. This produces the following goal.

```
Subgoal *1.2/4'5'
(IMPLIES (AND (CONSP (CONS TUPLE7 TUPLE8))
              (INTEGERP TUPLE7)
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT TUPLE8))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

This simplifies, using primitive type reasoning, to

```
Subgoal *1.2/4'6'
(IMPLIES (AND (INTEGERP TUPLE7)
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3))
         (EQUAL (FAC (+ 1 TUPLE3))
```

```
                (+ I (* I TUPLE3)))).
```

We suspect that the term (INTEGERP TUPLE7) is irrelevant to the truth of this conjecture and throw it out. We will thus try to prove

```
Subgoal *1.2/4'7'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

Name the formula above *1.2.1.

```
Subgoal *1.2/3
(IMPLIES (AND (NOT (ATOM TUPLE6))
              (INTEGERP (CAR TUPLE6))
              (NOT (CONSP (CDR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (INTEGER-LISTP TUPLE6)
              (CONSP TUPLE6))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

By the simple :definition ATOM we reduce the conjecture to

```
Subgoal *1.2/3'
(IMPLIES (AND (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6))
              (NOT (CONSP (CDR TUPLE6)))
              (INTEGERP I)
```

```
            (< 0 I)
            (INTEGERP TUPLE3)
            (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
            (INTEGER-LISTP TUPLE6))
       (EQUAL (FAC (+ 1 TUPLE3))
              (+ I (* I TUPLE3)))).
```

This simplifies, using the :definitions INTEGER-LISTP and MV-NTH, the :executable-counterparts of BINARY-+, INTEGERP, MV-NTH and ZP, primitive type reasoning, the :forward-chaining rules INTEGER-LISTP-FORWARD-TO-RATIONAL-LISTP and RATIONAL-LISTP-FORWARD-TO-TRUE-LISTP and the :type-prescription rules INTEGER-LISTP and RATIONAL-LISTP, to

```
Subgoal *1.2/3''
(IMPLIES (AND (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6))
              (NOT (CONSP (CDR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (INTEGER-LISTP (CDR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

This simplifies, using the :definition INTEGER-LISTP and primitive type reasoning, to

```
Subgoal *1.2/3'''
(IMPLIES (AND (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6))
              (NOT (CONSP (CDR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
```

```
            (NOT (CDR TUPLE6)))
          (EQUAL (FAC (+ 1 TUPLE3))
                 (+ I (* I TUPLE3)))).
```

This simplifies, using the :executable-counterpart of CONSP, to

```
Subgoal *1.2/3'4'
(IMPLIES (AND (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (CDR TUPLE6)))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

The destructor terms (CAR TUPLE6) and (CDR TUPLE6) can be eliminated by using CAR-CDR-ELIM to replace TUPLE6 by (CONS TUPLE7 TUPLE8), generalizing (CAR TUPLE6) to TUPLE7 and (CDR TUPLE6) to TUPLE8. This produces the following goal.

```
Subgoal *1.2/3'5'
(IMPLIES (AND (CONSP (CONS TUPLE7 TUPLE8))
              (INTEGERP TUPLE7)
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT TUPLE8))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

This simplifies, using primitive type reasoning, to

```
Subgoal *1.2/3'6'
```

```
(IMPLIES (AND (INTEGERP TUPLE7)
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

We suspect that the term (INTEGERP TUPLE7) is irrelevant to the truth of this conjecture and throw it out. We will thus try to prove

```
Subgoal *1.2/3'7'
(IMPLIES (AND (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

Name the formula above *1.2.2.

```
Subgoal *1.2/2
(IMPLIES (AND (NOT (ATOM TUPLE6))
              (INTEGERP (CAR TUPLE6))
              (NOT (INTEGER-LISTP (CDR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (INTEGER-LISTP TUPLE6)
              (CONSP TUPLE6))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).
```

By the simple :definition ATOM we reduce the conjecture to

```
Subgoal *1.2/2'
(IMPLIES (AND (CONSP TUPLE6)
              (INTEGERP (CAR TUPLE6))
              (NOT (INTEGER-LISTP (CDR TUPLE6)))
              (INTEGERP I)
              (< 0 I)
              (INTEGERP TUPLE3)
              (NOT (INTEGERP (MV-NTH 1 TUPLE6)))
              (INTEGER-LISTP TUPLE6))
         (EQUAL (FAC (+ 1 TUPLE3))
                (+ I (* I TUPLE3)))).

.
.
.

Q.E.D.

Summary
Form:  ( DEFTHM CORREC2 ...)
Rules: ((:DEFINITION AFAC3)
        (:DEFINITION AFAC2)
        (:DEFINITION MV-NTH)
        .
        .
        .
        (:EXECUTABLE-COUNTERPART BINARY-+)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART FAC)
        (:EXECUTABLE-COUNTERPART ZP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS))
Warnings:  Non-rec and Free
```

```
Time:  27.32 seconds ...
 CORREC2
```

Appendix B

Implementation of SMP algorithm

In the appendix, we produce some key parts of our implemtation of the SMP algorithm. We chose Perl for our implementation, because of its regular expression processing power.

B.1 Top Level

```
################### SMP ALGORITHM ################################
# SMP(Code)
#
#       If Is_Empty(Code)
#               Return ProgVariables
#
#       Else
#               1. LastStatement = GetLastStatement(Code)
#               2. LastFunction  = Process(LastStatement)
#               3. FirstFunction = SMP(Code without LastStatement)
#               4. Return Compose(LastFunction, FirstFunction)
##################################################################

################################## SMP ######################
```

```
#Argument is Code Block
sub SMP
{
        my $Return;
        my $CodeBody = $_[0];

        if (Is_Empty($CodeBody))
        {
                $Return = join("," , @ProgVariables);
        }
        else
        {
                my @tempArray     = GetLastStatement($CodeBody);
                my $CodeFirstPart = @tempArray[0];
                my $LastStatement = @tempArray[1];
                my $LastFunctionName = Process($LastStatement);
                $Return = $LastFunctionName."(".SMP("{".$CodeFirstPart."}")
                                                                        .")";
        }
        $Return;
}
```

B.2 Code for the Top Level Process Implementation

```
#########################################################################
#Process(Statement)
#
#       If Is_Assignment(Statement)
#                       ProcessAssignment(Statement)
#       If Is_While(Statement)
#                        ProcessWhile(Statement)
#       If Is_If(Statement)
```

```
#                       ProcessIf(Statement)
#
#####################################################################
sub Process
{
        my $Statement = $_[0];
        my $Return;

                ##################################
                #Is It Assignment
                ##################################
                if($Statement =~ m/^\s*(\[\d+\].*;)\s*$/)
                {
                        $Statement = $1;
                        $Return = ProcessAssignment($Statement);
                }

                #Is It While
                elsif($Statement =~ m/^\s*(while.*\})\s*$/)
                {
                        $Statement =~ $1;
                        $Return = ProcessWhile($Statement);
                }

                #Is It If
                elsif($Statement =~ m/^\s*(if.*\})\s*$/)
                {
                        $Statement =~ $1;
                        $Return = ProcessIf($Statement);
                }

                else
                {
                        $Return = $Statement
```

```
                                        ."==>Formatting Error (Process)";
                }
        $Return;
}
```

B.3 Code for the Process Assignment Implementation

```
my @AssignmentFunctionVariables;
my @AssignmentFunctions;
################## ProcessAssignment(AssignmentStatement)#############
#############################################################################
#Processes Assignment
#############################################################################
sub ProcessAssignment
{
        my $AssignmentStatement = $_[0];
        my $Return;

        ################################
        #Is It Really an Assignment ?
        ################################
        if($AssignmentStatement =~ m/^\s*(\[\d+\].*;)\s*$/)
        {
                $AssignmentStatement = $1;
                $Return = MakeAssignment($AssignmentStatement);
        }
        else
        {
                $Return = $AssignmentStatement
                        ."==>Formatting Error (ProcessAssignment)";
        }
        $Return;
```

```
}

#############################################################################
#       MakeAssignment(Statement)
#- Updates Model with Assignment Function Definition
#- Returns the name of the created function
#############################################################################
sub MakeAssignment
{
my $Return;
my $AssignmentStatement = $_[0];
my @FuncBody = @ProgVariables;
my $Funcdef;
        my $FuncInfo;

        ################################
        #Is It Really an Assignment ?
        ################################
        if($AssignmentStatement =~ m/^\s*(\[\d+\].*;)\s*$/)
        {
                $AssignmentStatement = $1;
                $AssignmentStatement =~ m/\[(\d+)\]\s*(.*);/;

                ######################################################
                #Create Function Name
                ######################################################
                $Return = "A".$Funcname.$1;

                $Funcdef = "\n\n\nDefining : \$".$Return
                            ."\$ from assignment statement :-\\emph\{\$"
                            .$AssignmentStatement."\$\}-\\\\\n";
```

```
$AssignmentStatement =~ m/^\[(\d+)\](\w_\d)(\s*)=(\s*)(.*);(\s*)$/;
@FuncBody = &replace($2, "(".$5.")", @FuncBody);

        ########################################################
        #Creating Function definitions
        ########################################################
        $Funcdef = $Funcdef."\$\\forall("
                    .join("," , @ProgVariables).")":";
        $Funcdef = $Funcdef."(".$Return
                    ."(".join("," , @ProgVariables)
                    .join("," , @ProgConstants).") = (";
        $Funcdef = $Funcdef.join("," , @FuncBody)
                    .join("," , @ProgConstants).")\$\\\\\";

        $FuncInfo = "\$\\forall(".join("," , @ProgVariables).")":";
        $FuncInfo = $FuncInfo."(\\exists"
                        ."(".join(",",@ProgVariablesSub)
                        .")":((".join(",",@ProgVariablesSub).")=";
        $FuncInfo = $FuncInfo.$Return."(".join("," , @ProgVariables)
                        .join("," , @ProgConstants).")))\$";

        ########################################################
        #Update Model
        ########################################################
        $Model = $Model.$Funcdef."\\\\\n";
        $Model = $Model.$FuncInfo."\\\\\n";
        $Model = $Model."End Definition \$("
                    .$Return.")\$\\\\\n\\hline\n\n";

}
else
{
        $Return = $AssignmentStatement
                    ."==>Formatting Error (MakeAssignment)";
```

```
        }
        $Return;
}
```

B.4 Code for the ProcessIf implementation

```
#Argument is an if Branch
sub ProcessIf
{
        my $IfStatement = @_[0];
        my $Cond;
        my $IfBody;
        my $ElseBody;
        my $Return = "Fail";

        #Is Really If Statement
        if($IfStatement =~ m/^\s*(if.*else.*\})\s*$/)
        {
                $IfStatement = $1;

                $IfStatement =~ m/^(if.*)(else.*)$/;

                my @Temp156 = ProcessIfHelper($1, $2);

                my $TempIfPart = @Temp156[0];
                my $TempElsePart = @Temp156[1];

                $TempIfPart =~ m/^\s*if\s*(\(.*?\))\s*(.*)$/;
                $Cond = $1;
                $IfBody = $2;

                $TempElsePart =~ m/^\s*else\s*(\{.*\})/;

                    $ElseBody = $1;
```

```
                print "\n***************".$ElseBody;
            $Return = MakeIfStatement($Cond, SMP($IfBody),
                        SMP($ElseBody));
        }
        $Return;
}



sub ProcessIfHelper
#First argument is If part of If/Else statement
#Second Argument is smallest possible Else part
#Returns the if part and the else part in an array
{
        my $FirstPart = @_[0];
        my $LastPart  = @_[1];
        my @Return;

        my @temp1 = $LastPart =~ /\{/g;
        my $NumOpenedBrackets = scalar @temp1;

        my @temp2 = $LastPart =~ /\}/g;
        my $NumClosedBrackets = scalar @temp2;


        if($NumOpenedBrackets == $NumClosedBrackets)
        {
                @Return = ($FirstPart, $LastPart);
        }
        elsif($NumOpenedBrackets < $NumClosedBrackets)
        {
                $FirstPart =~ m/(.*)(else.*?)$/;
                my $NewFirstPart = $1;
                my $NewLastPart = $2.$LastPart;
```

```
                @Return = ProcessIfHelper($NewFirstPart, $NewLastPart);
        }
        else
        {
                #Shouldn't happen
                @Return = ("Formatting Error (ProcessIfHelper)"
                            , "Formatting Error (ProcessIfHelper)");
        }
        @Return;
}


#MakeIfStatement($Cond, SMP($IfBody), SMP($ElseBody));
sub MakeIfStatement
{
    my $Cond = @_[0];
    my $IfBodyFunc = @_[1];
    my $ElseBodyFunc = @_[2];
    my $Return = $Cond.$IfBodyFunc.$ElseBodyFunc;
    $Return;
}
```

B.5 Code for the ProcessWhile implementation

```
#Argument is a while statement
sub ProcessWhile
{
        my $WhileStatement = @_[0];
        my $Cond;
        my $WhileBody;
        my $Return;
```

```
        #Is Really While Statement
        if($WhileStatement =~ m/^\s*(while.*\})\s*$/)
        {
                $WhileStatement = $1;
                $WhileStatement =~ m/^while\s*(\(.*?\))\s*(\{.*\})$/;
                $Cond = $1;
                $WhileBody = $2;
                $Return = MakeWhileStatement($Cond, SMP($WhileBody));
        }
        $Return;
}

my $WhileLoopNum = 0;
my $WhileLoopConsts = "";

sub MakeWhileStatement
#Arg 1 : Condition Statement
#Arg 2 : Name of the function that codifies the body of the while loop
{

        #NEEDS WORK
        my $CondStatement = shift @_;
        my $BodyCodeName  = shift @_;

        my $StdArgs = join("," , @ProgVariables).join("," , @ProgConstants);

        $WhileLoopNum++;
        my $Return = "W".$Funcname.$WhileLoopNum;
        my $WhileLoopConst = "l"."_".$WhileLoopNum;
        $WhileLoopConsts = $WhileLoopConsts." ".$WhileLoopConst;

        ######################################################################
        #While A Function (General function in partial completness)
        ######################################################################
```

```
my $WFuncAName = $Return;


my $WFuncADef = "\n\n\nDefining : \$".$Return
                ."\$ from while statement :-\\emph\{while\$"
                .$CondStatement."\\\{\dots\\\}\$\}-\\\\\n";


$WFuncADef = $WFuncADef."\$\\forall(".join("," , @ProgVariables)
                .",w):(";
$WFuncADef = $WFuncADef."(0 = w)\\implies(".$WFuncAName
                ."(".$StdArgs.",w) = (".$StdArgs.")))\$\\\\\n";
$WFuncADef = $WFuncADef."\$\\forall(".join("," , @ProgVariables)
                .",w):(";
$WFuncADef = $WFuncADef."(0 < w)\\implies(".$WFuncAName
                ."(".$StdArgs.",w) = ";

#Quick Hack
my $TempArgs = $WFuncAName."(".$StdArgs.",(w - 1))";
$BodyCodeNameTemp = $BodyCodeName;
$BodyCodeNameTemp =~ s/\((\w+,?)+?\)/\($TempArgs\)/;
$WFuncADef = $WFuncADef.$BodyCodeNameTemp."))\$\\\\\\\\\n\n";
$Model = $Model.$WFuncADef;



my $WFuncAInfo = "\$\\forall(".join("," , @ProgVariables)
                    .",w):(\\exists("
                    .join(",",@ProgVariablesSub).("):";
$WFuncAInfo = $WFuncAInfo."((".join(",",@ProgVariablesSub).")="
                    .$WFuncAName."(".$StdArgs.",w)))\$\\\\\n";

$Model = $Model.$WFuncAInfo;
$Model = $Model."End Definition \$("
```

```
                .$Return.")\$\\\\\n\\hline\n\n";


        ######################################################################
        #While B Function (Completness of loop) (Future Work)
        ######################################################################
        #my $WFuncBName = $Return."B";
        #my $WFuncBDef = "\\forall(".join("," , @ProgVariables)."):(";
        #$WFuncBDef = $WFuncBDef."\\neg".$CondStatement."\\implies("
                        .$WFuncBName."(".$StdArgs.") = 0))\n";
        #$BodyCodeNameTemp = $BodyCodeName;
        #$BodyCodeNameTemp =~ s/\((\w+,?)+?\)/\($StdArgs\)/;
        #$WFuncBDef = $WFuncBDef."\\forall(".join("," , @ProgVariables)
                        .")":(".$CondStatement."\\implies(".$WFuncBName
                        ."(".$StdArgs.") = 1 + (".$WFuncBName
                        ."(".$BodyCodeNameTemp.")))\n";
        #$Model = $Model.$WFuncBDef."\n";
        ######################################################################


        $Return;
}
```