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

UMI®
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 fulfilment 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

© Franck J.L. Binard, Ottawa, Canada, 2002
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

0-612-76565-2
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

1 Introduction .............................................. 1
  1.1 Problem Statement ................................... 1
  1.2 Approach and Motivation ............................. 1
    1.2.1 Current Approach ................................. 1
    1.2.2 Intrinsic Problems in Traditional Correctness Proof
            Techniques ......................................... 3
    1.2.3 Description of the Approach ..................... 8
  1.3 Contribution of the Thesis .......................... 12
  1.4 The Thesis Organization ............................. 12

2 Background and Function Set ............................ 14
  2.1 Recursive Functions .................................. 14
    2.1.1 Some Recursive Functions ....................... 16
  2.2 Tuple-Properties ..................................... 18

3 The Cc Programming Language ............................ 20
  3.1 Syntax ................................................ 20
  3.2 Hoare Triple Logic for Cc ............................ 23
    3.2.1 Hoare Triple Logic Rules for Cc.................. 23
  3.3 Computing Recursive Functions in Cc ................ 24

4 The Sl Logic ............................................. 40
  4.1 Sl, the Formal Language Overview ................... 40
  4.2 Syntax ................................................ 41
    4.2.1 Regular Language Operators: ..................... 41
5 Building $S_L$ Theories from $C^c$ Code Segments
5.1 Algorithm ........................................ 61
  5.1.1 Overview ........................................... 61
  5.1.2 Algorithm's Description ......................... 67
  5.1.3 ProcessAssignment .................................. 68
  5.1.4 ProcessIf .......................................... 72
  5.1.5 ProcessWhile ...................................... 79
  5.1.6 Justification for the Last Sentence in the Process Functions ................................................... 86
  5.1.7 Complexity Analysis .............................. 87

6 Implementations and Results ................................ 88
  6.1 The Nivea Implementation .............................. 88
  6.2 Commands ............................................. 89
    6.2.1 'GetProgram FilePath [FileName]' .................. 89
    6.2.2 'NormalizeProgram FileName' ....................... 89
    6.2.3 'Modelize ProgName' ................................ 91
  6.3 Experimental Results .................................. 92
  6.4 The ACL2 Experiments ................................ 104
    6.4.1 ACL2 Vocabulary and Background .................... 104
    6.4.2 Experiments ........................................ 108
    6.4.3 General Discussion of the Results of the ACL2 Experiments .................................................. 120

7 Conclusions ............................................ 121
  7.1 Summary of Contributions ............................ 121
  7.2 Comparisons with Existing Work ...................... 122
    7.2.1 Verification of Flow Chart Programs .............. 122
    7.2.2 Hoare Triple Logic ............................... 125
7.3 Future Work ........................................ 127
  7.3.1 Completeness .................................. 127
  7.3.2 $C^c$ Language Extensions ................. 128

A ACL2 output for the main proof of Chapter 6 131

B Implementation of SMP algorithm .............. 151
  B.1 Top Level ....................................... 151
  B.2 Code for the Top Level Process Implementation .... 152
  B.3 Code for the Process Assignment Implementation .... 154
  B.4 Code for the ProcessIf implementation ............ 157
  B.5 Code for the ProcessWhile implementation ..... 159
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 $\phi$ and $\varphi$ 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 $\phi$ can be proven to hold before the execution of $PrgStruct$ (precondition), then the statement $\varphi$ can be proven to hold immediately after the execution of $PrgStruct$ (post-condition).

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

- $\frac{[\phi]}{[\phi]}$ \text{FUNC1}  \frac{[\eta]}{[\eta]}$\text{FUNC2} [\psi]}{[\psi]}$ \text{COMPOSITION}

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

- $\frac{[\phi \land \beta]}{[\phi]} \frac{[\psi]}{[\psi]}$ \text{FUNC1}  \frac{[\varphi \land \neg \beta]}{[\varphi]} \text{FUNC2} [\psi]}{[\psi]}$ \text{IF-STATEMENT}

- $\frac{[\psi \land \beta]}{[\psi]}$ \text{FUNC1}  \frac{[\psi]}{[\psi]}$ \text{PARTIAL WHILE}
\[ \phi' \rightarrow \phi \quad (|\phi|) \quad \text{FUNC1} (|\psi|) \quad \psi \rightarrow \psi' \quad \text{IMPLIED} \]

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:

```c
def 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.
\[ n \geq 0 \] \quad \text{PRE CONDITION}

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

\[ n \geq 0 \]
\[ (n \geq 0) \land (x^0 = 1) \] \quad \text{COPY}
\[ (n \geq 0) \land (x^a = 1) \] \quad \text{IMPLIED}

int a = 0;

\[ (n \geq a) \land (x^a = 1) \] \quad \text{ASSIGNMENT}

int b = 1;

\[ (n \geq a) \land (x^a = b) \] \quad \text{ASSIGNMENT}

int c = 1;

\[ (n \geq a) \land (x^a = b) \] \quad \text{ASSIGNMENT}

while(\!n > a){

\[ (n \geq a) \land (n \geq a) \land (x^a = b) \] \quad \text{HYP and INV}
\[ (n \geq a) \land (x^a = b) \] \quad \text{IMPLIED}
\[ (n \geq a + 1) \land (x^{a+1} = x \ast b) \] \quad \text{IMPLIED}

a = a + 1;

\[ (n \geq a) \land (x^a = x \ast b) \] \quad \text{ASSIGNMENT}

b = x \ast b;

\[ (n \geq a) \land (x^a = b) \] \quad \text{ASSIGNMENT}

c = c \ast a;

\[ (n \geq a) \land (x^a = b) \] \quad \text{ASSIGNMENT}
```
\[
\neg(n > a) \land (n \geq a) \land (x^n = b) \quad \text{PARTIAL WHILE}
\]
\[
(n = a) \land (x^n = b) \quad \text{IMPLIED}
\]
\[
x^n = b \quad \text{IMPLIED}
\]

\[
x^n = b \quad \text{POST CONDITION}
\]

Consider now the following proof on the same program:

\[
 n \geq 0 \quad \text{PRE CONDITION}
\]

```c
void func (int x, int n){
    
    int a = 0;
    
    int b = 1;
    
    int c = 1;
    
    while(n > a){
        
        \[
        (n \geq a) \land (a! = c) \quad \text{ASSEIGNMENT}
        \]
    }
}
```

\[
 n \geq 0 \quad \text{COPY}
\]
\[
(n \geq 0) \land (0! = 1) \quad \text{IMPLIED}
\]

```c
int a = 0;

int b = 1;

int c = 1;

while(n > a){
    
    \[
    (n \geq a) \land (a! = c) \quad \text{ASSEIGNMENT}
    \]
}
```
\[ a = a + 1; \]
\[ \lfloor (n \geq a) \land (a! = c \ast a) \rfloor \text{ ASSIGNMENT} \]
\[ b = x \ast b; \]
\[ \lfloor (n \geq a) \land (a! = c \ast a) \rfloor \text{ ASSIGNMENT} \]
\[ c = c \ast a; \]
\[ \lfloor (n \geq a) \land (a! = c) \rfloor \text{ ASSIGNMENT} \]
\} 
\[ \neg(n > a) \land (n \geq a) \land (a! = c) \rfloor \text{ PARTIAL WHILE} \]
\[ (n = a) \land (a! = c) \]
\[ (n! = c) \]
\[ \text{ IMPLIED} \]
\[ \text{ IMPLIED} \]
\} 
\[ (n! = c) \rfloor \text{ POST CONDITION} \]

While in the first proof, \( \lfloor (n \geq a) \land (x^a = b) \rfloor \) is chosen as the invariant to the while loop, \( \lfloor (n \geq a) \land (a! = c) \rfloor \) 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 $\land$ operator of the chosen logic (for example, we could use $| (n \geq a) \land (a! = c) \land (x^n = b)|$ 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 $\land$ 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:

<table>
<thead>
<tr>
<th>(x, n)</th>
<th>(a, b, c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>(1, 0, 1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>(1, 1, 1)</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>(2, 1, 2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(2, 0)</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>(1, 2, 1)</td>
</tr>
<tr>
<td>(2, 2)</td>
<td>(2, 4, 2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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()`:

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

1. It can be considered to be equivalent to the function \( FA : \mathbb{N}^3 \rightarrow \mathbb{N}^3 \) 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:

\[
[(u = a) \land (v = b) \land (w = c)][funcA()](a, b, c) = FA(u, v, w)]
\]

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:
\[
\forall a, b, c, u, v, w \in \mathbb{N}:
((u, v, w) = FA(a, b, c)) \land (a! = c) \implies (u! = w)
\]
– Notice (and prove) that \( FA \) satisfies the following property:
\[
\forall a, b, c, u, v, w \in \mathbb{N}:
((u, v, w) = FA(a, b, c)) \land (x^a = b) \implies (x^u = v)
\]

Here, the symbol "\( \implies \)" is used as an abbreviation 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 \( FB : \mathbb{N}^3 \to \mathbb{N}^3 \) defined as:
\[
FB(u, v, w) = (0, 1, 1)
\]

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

• We would now look at the set of triples:
\[
S = \{ t : \mathbb{N}^3 | (x, y, z) = FA^m \circ FB(0, 0, 0), (0 \leq m) \}
\]

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 \(\text{void func}(\text{int } x, \text{ int } n)\), if when it finishes executing,

\[
|T\{\text{func}\}(a_e = a) \land (b_e = b) \land (c_e = c)|
\]

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

**Definition 2.1.1 (Primitive Recursion[1])** Let $p \in \mathbb{N}$, $g \in \mathbb{F}_p$ 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. $f(x_1, x_2, \ldots, x_p, 0) = g(x_1, x_2, \ldots, x_p)$
2. $f(x_1, \ldots, x_p, y + 1) = h(x_1, \ldots, x_p, y, f(x_1, \ldots, x_p, y))$
Definition 2.1.2 ($\mu$-recursion[2]) Let $p \in \mathbb{N}$ be fixed. A function $f \in \mathbb{F}_p$ is defined from a relation $R$ by $\mu$-recursion if:

1. $\forall (x_1, \ldots, x_p) \in \mathbb{N}^p, \exists y \in \mathbb{N}$ such that $R(x_1, \ldots, x_p, y)$

2. $f(x_1, \ldots, x_p) = \mu y R(x_1, \ldots, x_p, y)$ where $\mu y R(x_1, \ldots, x_p, y)$ is the least number $y$ such that $R(x_1, \ldots, x_p, y)$ holds.

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

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

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) $\circ(x) = 0$

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

   (iii) $\partial^p_i (x_1, \ldots, x_n) = x_i$

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

3. Closed under primitive recursion and $\mu$-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 $\mu$-recursion.

Definition 2.1.4 (Characteristic functions) Let $A \subseteq \mathbb{N}^p$. The characteristic function $\chi_A : \mathbb{N}^p \rightarrow \{0, 1\}$ 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 annotated 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 \( \text{Add}(x, y) \).

2. We define subtraction in two steps:

   (i) Defining \( \text{Pred}(x) \) by primitive recursion as:

   \[
   \begin{align*}
   \text{Pred}(0) &= 0 \\
   \text{Pred}(x + 1) &= x
   \end{align*}
   \]

   (ii) Allows us to define \( \cdot(x, y) \) as:

   \[
   \begin{align*}
   \cdot(x, 0) &= x \\
   \cdot(x, y + 1) &= \text{Pred}(\cdot(x, y))
   \end{align*}
   \]

   And of course, we will write \( x \cdot y \) for \( \cdot(x, y) \).

3. We define multiplication:

   \[
   \begin{align*}
   \text{Mult}(x, 0) &= 0 \\
   \text{Mult}(x, y + 1) &= \text{Add}(x, (\text{Mult}(x, y)))
   \end{align*}
   \]

   We write \( x \cdot y \) for \( \text{Mult}(x, y) \).

4. We define \( \text{IsnntZero}(x) \) as

   \[ \text{IsnntZero}(x) := \text{Sub}(1, \text{Sub}(1, x)) \]

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

   \[ \chi_{\text{Sm}}(x, y) := \text{IsnntZero}((\text{Sub}(y, x)) \]

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

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

   \[ \chi_{\text{Eq}}(x, y) := \text{Sub}(1, \text{IsnntZero}(\text{Add}((\text{Sub}(x, y), \text{Sub}(y, x)))) \]

   We write \( x = y \) to mean \( \chi_{\text{Eq}}(x, y) \)
7. We call a projection function \( P^q_p : \mathbb{N}^q \rightarrow \mathbb{N}^p \) (\( p \leq q \)) any function which can be defined as:

\[
\{(\partial^p_s(x_1, \ldots, x_q), \ldots, \partial^q_s(x_1, \ldots, x_q)) | (x_1, \ldots, x_q) \in \mathbb{N}^q, \forall i, s_i \in \mathbb{N}, s_i < s_{i+1}, s_q \leq q\}
\]

Given \( p, q \in \mathbb{N} \) such that \( p \leq q \), a projection function \( P^q_p : \mathbb{N}^q \rightarrow \mathbb{N}^p \), a recursive set \( A \subseteq \mathbb{N}^p \), and 2 recursive functions \( f \) and \( g \in \mathbb{F}_q \), we define the function \( I f E l e x e \) corresponding to

\[
\begin{cases}
I f E l e x e(x_1, \ldots, x_q) = f(x_1, \ldots, x_q) & \text{if } P^q_p(x_1, \ldots, x_q) \in A \\
I f E l e x e(x_1, \ldots, x_q) = g(x_1, \ldots, x_q) & \text{otherwise}
\end{cases}
\]

by composition as:

\[
I f E l e x e(x_1, \ldots, x_q) := A d d(M u l t(f(x_1, \ldots, x_q), \chi_A(P^q_p(x_1, \ldots, x_q))) , M u l t(g(x_1, \ldots, x_q), S u b(1, \chi_A(P^q_p(x_1, \ldots, x_q)))))
\]

It is also be proven that the predicates \( \leq, >, \geq \), the functions \( E x p o n e n t i a t i o n, F a c t o r i a l \) and \( F i b o n n a c i \), and the boolean operators \( \wedge, \rightarrow, \forall, \neg \) 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 \( p \in \mathbb{N} \). A Tuple-Property is a pair of tuples :

\[
< t = (t_1, t_2, \ldots, t_p), F = (f_1, \ldots, f_p) >
\]

satisfying:

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

**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 \( f_1(x_1, x_2, x_3, x_4) = \partial^4_1(x_1, x_2, x_3, x_4) \)

- \( 2 = f_2(3, 2, 6, 9), \) where \( f_2(x_1, x_2, x_3, x_4) = \partial^4_2(x_1, x_2, x_3, x_4) \)
• $6 = f_3(3, 2, 6, 9)$, where $f_3(x_1, x_2, x_3, x_4) = \text{Fac}(\partial^1_3(x_1, x_2, x_3, x_4))$

• $9 = f_4(3, 2, 6, 9)$, where $f_4(x_1, x_2, x_3, x_4) = \text{Exp}(\partial^1_3(x_1, x_2, x_3, x_4), \partial^1_1(x_1, x_2, x_3, x_4))$

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

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

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):

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

Up to now, we haven't been very precise when talking about "code segments".
In the context of the \( C^c \) language, a code segment is a \textit{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:

```c
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:

```plaintext
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 (\( \langle \phi \rangle \) FUNC (\( \langle \psi \rangle \)) is satisfied under **partial correctness** if, for all states which satisfy \( \phi \) (\( \phi \) and \( \psi \) being some well formed expressions in a chosen logic), the state resulting from FUNC's execution satisfies the postcondition \( \psi \). provided that FUNC actually terminates. In this case, we write:

\[ \vdash_{\text{par}} (\langle \phi \rangle) \text{FUNC} (\langle \psi \rangle) \]

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 $p \in \mathbb{N}$ be fixed, $f \in \mathbb{F}_p$ a recursive function and $(n_1, n_2, ..., n_p)$ a fixed tuple in $\mathbb{N}^p$. 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:

   $$\begin{array}{c}
   \top \ | \ \text{PRE CONDITION} \\
   C(n) \\
   \{
   [c] = [c] \ | \ \text{IMPLIED} \\
   x_1 = [c];
   \}
   \end{array}$$
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:

   \[\top\] \text{PRE CONDITION}

   ```
   s(n)
   {
     \[n = n\] \text{IMPLIED}
     \[n + 1 = n + 1\] \text{IMPLIED}
     x_1 = n;
   }
   ```

   \[x_1 + 1 = n + 1\] \text{ASSIGNMENT}

   ```
   x_1 = x_1 + 1;
   ```
\[ x_1 = n + 1 \] \hspace{1cm} \text{ASSIGNMENT}

\}

\[ x_1 = n + 1 \] \hspace{1cm} \text{COPY}

Provides the proof.

3. **Projection function**

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

```latex
\text{pi}(n1, n2, \ldots, np)
\{
    x_1 = [ni];
\}
```

Where \([ni]\) is replaced by the \(i\)th parameter which is passed on to the function.

The following proof pattern:

\[ \top \] \hspace{1cm} \text{PRE CONDITION}

\text{pi}(n1, n2, \ldots, np)
\{
\}

\[ n_i = n_i \] \hspace{1cm} \text{IMPLIED}

\hspace{1cm} x_1 = [ni];

\[ x_1 = n_i \] \hspace{1cm} \text{ASSIGNMENT}

\}

26
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 occurrences 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 $x_i$ 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:

\[
\text{COMP}(n_1, n_2, \ldots, n_p)
\{
    x_1 = 0;

    [\text{body of JAY1 running on input } (n_1, \ldots, n_p) \text{ with } ]
    [\text{local variables } (x_1, x_2, \ldots, x_{[G1]}) ]
    [\text{reindexed to } (x_2, x_3, \ldots, 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

28
\[ x_{1} = 0; \]

\[ |x_{1} = 0| \] ASSIGNMENT
\[ |T| \]

[body of JAY1 running on input $(n_{1}, \ldots, n_{p})$ with]
[JAY1 local variables $(x_{1}, x_{2}, \ldots, x_{\{G1\}})$]
[reindexed to $(x_{2}, x_{3}, \ldots, x_{\{G1 + 1\}})$]

\[ |x_{2} = g_{1}(n_{1}, \ldots, n_{p})| \] (From Induction Hyp)

\[ |(x_{1} = 0) \land (x_{2} = g_{1}(n_{1}, \ldots, n_{p}))| \] IMPLIED
\[ |T| \]

[body of JAY2 running on input $(n_{1}, \ldots, n_{p})$ with]
[local variables $(x_{1}, x_{2}, \ldots, x_{\{G2\}})$]
[reindexed to $(x_{\{G1 + 2\}}, x_{\{G1 + 3\}}, \ldots, x_{\{G1 + G2 + 2\}})$]

\[ |x_{\{G1 + 2\}} = g_{2}(n_{1}, \ldots, n_{p})| \] (From Induction Hyp)

\[ |(x_{1} = 0) \land (x_{2} = g_{1}(n_{1}, \ldots, n_{p})) \land (x_{\{G1 + 2\}} = g_{2}(n_{1}, \ldots, n_{p}))| \] IMPLIED

\[ |T| \]
[body of JAYk running on input \((n_1, \ldots, n_p)\) with ]
[local variables \((x_1, x_2, \ldots, x_{[Gk]}\)) reindexed to ]
\[\{(x_{[G1 + G2 + \ldots + G(k-1) + p]}, x_{[G1 + G2 + \ldots + G(k-1) + p + 1]}, \]
\[\ldots, x_{[G1 + G2 + \ldots + G(k-1) + Gk + p]}\}\]
\[|x_{[G1+G2+\ldots+G(k-1)+p]} = g_k(n_1, \ldots, n_p)| \text{ (From Induction Hyp)}\]

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

\[|T|\]

[body of ASH running on input ]
\[\{(x_2, x_{[G1 + 2]}, x_{[G1 + G2 + 3]}, \ldots, \]
\[x_{[G1 + G2 + \ldots + G(k-1) + p]}\}\]
[with local variables \((x_1, x_2, \ldots, x_{[H]}\)) reindexed to ]
\[\{(x_{[G1 + G2 + \ldots + G(k-1) + Gk + p + 1]}, \ldots, \]
\[x_{[G1 + G2 + \ldots + G(k-1) + Gk + 2]}\}\]
\[|x_{[G1+G2+\ldots+G(k-1)+Gk+p+1]} = h((x_2, x_{[G1+2]}, x_{[G1+G2+3]}, \ldots, x_{[G1+G2+\ldots+G(k-1)+p]}))|\]

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

30
\[
x_{-1} = x_{-}[G1 + G2 + \ldots + G(k-1) + Gk + p + 1];
\]

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, \ldots, np, y)**

\[
\begin{align*}
  &x_1 = 0; \\
  &x_2 = 0; \\
  &\text{[Body of JAY running on input (n1, n2, \ldots, np) with JAY local]} \\
  &\text{[variables (x_1, \ldots, x_[G] reindexed to (x_3, \ldots x_[G + 2])]} \\
  &\text{[in REC]} \\
  &x_{-1} = x_3; \\
  &\text{while(x_2 < y)} \\
  &\text{\{x_2 = x_2 + 1;} \\
  &\text{[Body of ASH running on input (n1, n2, \ldots, np, x_2-1, x_1)]} \\
  &\text{[with ASH local variables (x_1, \ldots, x_[H]) reindexed to]} \\
\end{align*}
\]
\[ x_{[G+3]}, \ldots x_{[G+3+H]} \text{ in REC} \]

\[ x_{1} = x_{[G+3]}; \]

\}

\}

**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:

<table>
<thead>
<tr>
<th>( T )</th>
<th>PRE CONDITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{REC(n1, n2, \ldots, np, y)} )</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>( f(n_1, n_2, \ldots, n_p, 0) = g(n_1, n_2, \ldots, n_p) )</td>
<td>IMPLIED</td>
</tr>
<tr>
<td>( x_{1} = 0; )</td>
<td></td>
</tr>
<tr>
<td>( f(n_1, n_2, \ldots, n_p, 0) = g(n_1, n_2, \ldots, n_p) )</td>
<td>IMPLIED</td>
</tr>
<tr>
<td>( x_{2} = 0; )</td>
<td></td>
</tr>
<tr>
<td>( f(n_1, n_2, \ldots, n_p, x_2) = g(n_1, n_2, \ldots, n_p) )</td>
<td>ASSIGNMENT</td>
</tr>
<tr>
<td>(</td>
<td>T</td>
</tr>
<tr>
<td>[Body of JAY running on input ( (n_1, n_2, \ldots, n_p) ) ]</td>
<td></td>
</tr>
<tr>
<td>[with local variables ( x_1, \ldots, x_{[G]} ) reindexed ]</td>
<td></td>
</tr>
<tr>
<td>[to ( x_{3}, \ldots x_{[G+2]} ) ]</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>x_3 = g(n_1, n_2, \ldots, n_p)) (By Induction Hyp)</td>
</tr>
</tbody>
</table>

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

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

32
\( x_1 = x_3; \)

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1)| \] \text{ ASSIGNMENT}

\textbf{while} (x_2 < y)

\{ 

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \land (x_2 < y)| \] \text{ INARIANT HYP and GUARD}

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \land (f(n_1, n_2, \ldots, n_p, x_2 + 1) = h(n_1, n_2, \ldots, n_p, x_2, f(n_1, n_2, \ldots, n_p, x_2))| \] \text{ IMPLIED}

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \land (f(n_1, n_2, \ldots, n_p, x_2 + 1) = h(n_1, n_2, \ldots, n_p, x_2, x_1)| \] \text{ IMPLIED}

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \land (f(n_1, n_2, \ldots, n_p, x_2 + 1) = h(n_1, n_2, \ldots, n_p, x_2 + 1 - 1, x_1)| \] \text{ IMPLIED}

\( x_2 = x_2 + 1; \)

\[ |f(n_1, n_2, \ldots, n_p, x_2) = h(n_1, n_2, \ldots, n_p, x_2 - 1, x_1)| \] \text{ ASSIGNMENT}

\[ |T| \]

[Body of ASH running on input \((n_1, n_2, \ldots, n_p, x_2-1, x_1)\) \]

[with local ASH variables \((x_1, \ldots, x[H])\) reindexed to \]

[\(x_{[G+3]}, \ldots, x_{[G+3+H]}\)]

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

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = h(n_1, n_2, \ldots, n_p, x_2 - 1, x_1)) \land (x_{[G+3]} = h(n_1, n_2, \ldots, n_p, x_2 - 1, x_1))| \] \text{ IMPLIED}

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_{[G+3]})| \] \text{ IMPLIED}

33
\[ x_1 = x_\text{[G + 3]}; \]

\[ |f(n_1, n_2, \ldots, n_p, x_2) = x_1| \quad \text{ASSIGNMENT} \]

\}

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \land (x_2 \leq y) \land \neg(x_2 < y)| \quad \text{PARTIAL-WHILE} \]

\[ |(f(n_1, n_2, \ldots, n_p, x_2) = x_1) \land (x_2 = y)| \quad \text{IMPLIED} \]

\[ |f(n_1, n_2, \ldots, n_p, y) = x_1| \quad \text{IMPLIED} \]

\[ |x_1 = f(n_1, n_2, \ldots, n_p, y)| \quad \text{IMPLIED} \]

\}

\[ |x_1 = f(n_1, n_2, \ldots, n_p, y)| \quad \text{COPY} \]

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.

\[ \text{MREC}(n_1, n_2, \ldots, n_p) \]

{ \[ x_\text{.1} = 0; \]

\[ x_\text{.2} = 0; \]

[Body of JAY running on input]

[\((n_1, n_2, \ldots, n_p, x_\text{.1})\) with local]

[variables \((x_\text{.1}, \ldots, x_\text{[G]})\) reindexed]

[to \((x_\text{.3}, \ldots x_\text{[G + 2]})\)]

\[ x_\text{.2} = x_\text{.3}; \]

while(0 < x_\text{.2})

34
\{ 
\begin{align*}
    x_1 &= x_1 + 1; \\
    [\text{Body of JAY running on input}] \\
    [(n1, n2, \ldots, np, x_1) \text{ with local}] \\
    [\text{variables } (x_1, \ldots, x_{[G]}) \text{ reindexed}] \\
    [\text{to } (x_{[G + 3]}, \ldots, x_{[G + G + 3]})] \\
    x_2 &= x_{[G + 3]}; \\
\end{align*} \\
\}

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

\begin{tabular}{|l|}
\hline
\begin{tabular}{l}
\text{PRE CONDITION} \\
MREC(n1, n2, \ldots, np) \\
\end{tabular} \\
\hline
\end{tabular}

\[
[-\exists x \in \mathbb{N}(0 \leq x < 0)] \\
[\forall x \in \mathbb{N}(0 \leq x < 0) \rightarrow (g(n1, \ldots, n_p, x) \neq 0)] \\
\begin{array}{l}
\text{IMPLIED} \\
\text{IMPLIED} \\
\end{array}
\]

\[
x_1 = 0; \\
\]

\[
[\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n1, \ldots, n_p, x) \neq 0)] \\
\begin{array}{l}
\text{ASSIGNMENT} \\
\end{array}
\]

\[
x_2 = 0; \\
\]

\[
[\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n1, \ldots, n_p, x) \neq 0)] \\
\begin{array}{l}
\text{COPY} \\
\end{array}
\]

\[T]\]

\[
[\text{Body of JAY running on input}] \\
[(n1, n2, \ldots, np, x_1) \text{ with local}] \\
[\text{variables } (x_1, \ldots, x_{[G]}) \text{ reindexed}] \\
[\text{to } (x_3, \ldots, x_{[G + 2]})] \\
\]
\[ x_3 = g(n_1, n_2, \ldots, n_p, x_1) \] (From Induction Hyp)

\[
\begin{array}{l}
(x_3 = g(n_1, n_2, \ldots, n_p, x_1)) \\
\land (\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0))
\end{array}
\] IMPLIED

\[ x_2 = x_3; \]

\[
\begin{array}{l}
(x_2 = g(n_1, n_2, \ldots, n_p, x_1)) \\
\land (\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0))
\end{array}
\] ASSIGNMENT

\[
\text{while}(0 < x_2) \\
\{ \\
\}
\]

\[
\begin{array}{l}
(x_2 = g(n_1, n_2, \ldots, n_p, x_1)) \\
\land (\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0)) \\
\land (0 < x_2)
\end{array}
\] INVARIANT and GUARD

\[
\begin{array}{l}
(x_2 = g(n_1, n_2, \ldots, n_p, x_1)) \\
\land (\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0)) \\
\land (x_2 <> 0)
\end{array}
\] IMPLIED

\[
\begin{array}{l}
g(n_1, n_2, \ldots, n_p, x_1) <> 0 \\
\land (\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0))
\end{array}
\] IMPLIED

\[
\begin{array}{l}
\forall x \in \mathbb{N}(0 \leq x \leq x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0)
\end{array}
\] IMPLIED

\[
\begin{array}{l}
\forall x \in \mathbb{N}(0 \leq x < x_1 + 1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0)
\end{array}
\] IMPLIED

\[ x_1 = x_1 + 1; \]

\[
\begin{array}{l}
\forall x \in \mathbb{N}(0 \leq x < x_1) \rightarrow (g(n_1, \ldots, n_p, x) <> 0)
\end{array}
\] ASSIGNMENT

\[ |T| \]

36
\[ \text{[Body of JAY running on input]} \]
\[ \text{[(n1, n2, ..., np, x_1) with local]} \]
\[ \text{[variables (x_1, ..., x_{[G]} reindexed]} \]
\[ \text{to (x_{[G + 3]}, ..., x_{[G + G + 3]}]} \]
\[
\begin{align*}
(x_{[G + 3]} &= g(n_1, n_2, ..., n_p, x_1) \land (x_{[G + 3]} \geq 0) & \text{From Induction} \\
\forall x \in \mathbb{N}(0 \leq x < x_1) & \rightarrow (g(n_1, ..., n_p, x) \gg 0)) \\
X_2 &= X_{[G + 3]} ; \\
(x_2 &= g(n_1, n_2, ..., n_p, x_1) \land (x_2 \geq 0) & \text{IMPLIED} \\
\forall x \in \mathbb{N}(0 \leq x < x_1) & \rightarrow (g(n_1, ..., n_p, x) \gg 0)) \\
\text{]} \\
\end{align*}
\]
\[
\begin{align*}
(x_2 &= g(n_1, n_2, ..., n_p, x_1)) \land (x_2 \geq 0) & \text{PARTIAL-WHILE} \\
\forall x \in \mathbb{N}(0 \leq x < x_1) & \rightarrow (g(n_1, ..., n_p, x) \gg 0)) \\
\neg(0 < x_2) & \\
(x_2 &= g(n_1, n_2, ..., n_p, x_1)) \land (x_2 = 0) & \text{IMPLIED} \\
\forall x \in \mathbb{N}(0 \leq x < x_1) & \rightarrow (g(n_1, ..., n_p, x) \gg 0)) \\
\end{align*}
\]
\[
\begin{align*}
(g(n_1, n_2, ..., n_p, x_1) &= 0) & \text{IMPLIED} \\
\forall x \in \mathbb{N}(0 \leq x < x_1) & \rightarrow (g(n_1, ..., n_p, x) \gg 0)) \\
(x_1 &= \mu y(g(n_1, n_2, ..., n_p, y) = 0)) & \text{IMPLIED} \\
(x_1 &= f(n_1, n_2, ..., n_p)) & \text{IMPLIED} \\
\text{]} \\
\end{align*}
\]
\[ x_1 = f(n_1, n_2, \ldots, n_p) \]

\[ \Box \]

**Corollary 3.3.3 (C^e 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 \( C^e \) 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 \( C^e \) program using the TEE programs:

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

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

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

38
[Body of TEEp program running on input \((t_1, t_2, \ldots, t_p)\) with local variables \((x_1, x_2, \ldots, x_{[T1]})\) reindexed to \((x_{[p + T1 + \ldots + T(p-1) + 1]}, x_{[p + T1 + \ldots + T(p-1) + 2]}, \ldots, x_{[p + T1 + \ldots + T(p-1) + Tp]})\)]

\[
\begin{align*}
x_1 &= x_{[p + 1]}; \\
x_2 &= x_{[p + T1 + 1]}; \\
&\quad \ldots \\
x_p &= x_{[p + T1 + \ldots + T(p-1) + 1]}; \\
\end{align*}
\]

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

40
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 \( \Gamma = \{A, B, \ldots, Z, AA, AB, \ldots\} \)
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 \( \{\text{ind.}+, -, \cdot, (.), \leq, =, \lor, \land, \rightarrow, \forall\} \)

The symbol \( \text{ind.} \) 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 omitted 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

\[ A ::= x|\text{ind}|m|(A \ast A)|(A + A)|(A - A) \text{ where } x \in \chi, m \in \mathbb{N} \]

Examples:

'4', '\(x_1 - x_2\)', '\((y_1 + \text{ind}) - (x_2 + 2)\)' are examples of type A terms.

\[ B ::= (A[,\ A]^*) \]

Example:

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

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

Examples:

'\((y_1 \ast (5 + x_1)) \ast ((x_1 + x_2) \ast 2)\)'

'\text{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

• ‘\text{FUNC}(\text{EXP}((n + 1), 4), 4, (y_1 + 2))’

\text{FUNC}(\text{EXP}((n + 1), 4), 4, (y_1 + 2))
\text{FUNC}(\text{EXP}(A, A). A, A)
\text{FUNC}(\text{EXP}B, A, A)
\text{FUNC}(C)
D

• ‘\text{FUNC}(\text{APPL}((n+1), 4), 4, (x_1, x_2, x_1), \text{POL1}(x_2). \text{FUNC2}((y_1+2)))’

\text{FUNC}(\text{APPL}((n+1), 4), 4, (x_1, x_2, x_1), \text{POL1}(x_2). \text{FUNC2}((y_1+2)))
\text{FUNC}(\text{APPL}B. A, B, \text{POL1}B. \text{FUNC2}B)
\text{FUNC}(C)
D

• ‘\text{FUNC1}(\text{FUNC2}(\text{FUNC3}(x_1, x_2, x_3). n))’

\text{FUNC1}(\text{FUNC2}(\text{FUNC3}(x_1, x_2, x_3). n))
\text{FUNC1}(\text{FUNC2}(\text{FUNC3}(A, A, A). A))
\text{FUNC1}(\text{FUNC2}(\text{FUNC3}B. A))
\text{FUNC1}(\text{FUNC2}(C))
\text{FUNC1}(D)
D

43
Sentences of type E and F

\[ E ::= < D = D > \mid < A = A > \mid < A < A > \]

Examples:

'\( < (x_1, x_2, x_3) = \text{FUNC1}(\text{FUNC2}(0, y_1, 3)) > \)'

'\( < \text{FUNC1}(x_1) = \text{FUNC2}(x_2, x_3) > \)'

'\( < (x_1, x_2, x_3) = (x_1, x_2, x_3, x_4) > \)'

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 \lor F) | (F \land F) | (F \rightarrow F) | \forall(x) : F | \exists(x) : F \text{ where } x \in \}

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

Notation:

- Any valid expression matching \((F \land F) +^\uparrow \mid [F \lor F] +^\uparrow\) 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 $X, Y, Z$ with elements of $\Gamma$ and $n, m$ with elements of $\mathbb{N}$.

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. $S\{h \Rightarrow c\}$ 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 $\gamma$ is provable from $\Sigma$ if one of the following applies:

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

In that case, we write:

$$\Sigma \vdash \gamma$$

45
4.3.1 $S_L$ Inference Rules

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

  1. \[
  \frac{F_1 \quad F_2 \quad \ldots \quad F_p}{(F_1 \land F_2 \land \ldots F_p)} \quad (\land \text{ Intro})
  \]

  2. \[
  \frac{F_1 \land F_2 \land \ldots \land F_p}{F_i} \quad (\land \text{ Elim})
  \]
  
  Where ($1 \leq i \leq p$).

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

  1. \[
  \frac{F_i}{(F_1 \lor \ldots F_i \lor \ldots F_p)} \quad (\lor \text{ Intro})
  \]

  2. \[
  \frac{(F_1) \quad (F_2) \quad \ldots \quad (F_p)}{F_1 \lor F_2 \ldots \lor F_p \quad F \quad F \quad \ldots \quad F} \quad (\lor \text{ Elim})
  \]
  
  Where $F_1 \ldots F_p$ are Assumptions that are discharged by this rule’s application.

• $\rightarrow$ rules:
1. 

\[
\frac{F_2}{F_1 \rightarrow F_2} (\rightarrow \text{Intro })
\]

Where \( F_1 \) is an Assumption that is discharged by this rule's application.

2. 

\[
\frac{F_1 \rightarrow F_2}{F_2} F_1 (\rightarrow \text{Elim })
\]

• = Rules :

1. 

\[
< A_1 = A_1 > | < D_1 = D_1 > (\ = \text{Intro })
\]

2. 

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

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

3. 

\[
\frac{< #1 = #2 >}{F \{ x \rightarrow #1 \} \rightarrow \#2} (\ = \text{Sub})
\]

• ∀ rules :

1.

\[
\frac{F}{\forall(x) : F} (\forall \text{Intro })
\]

Where \( x \) is a variable that doesn't occur freely previously in the proof.
**Notation:**

Multiple applications of the $\forall$ Intro rule are written:

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

2.

$$
\frac{\forall(x) : F}{F \{x \Rightarrow A\} \quad (\forall \text{ Elim })}
$$

Note that it is now possible to derive:

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

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

- $\exists$ rules:

  1.

  $$
  \frac{(x_0) F \{x \Rightarrow x_0\}}{\exists(x) : F \quad F_1 \quad (\exists \text{ Elim })}
  $$

  where $x_0$ does not occur in $F_1$.

- Rename rule:

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

  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 \( \Sigma \) be defined as:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Premiss</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5</td>
<td>( \forall(x_1, x_2, x_3) : &lt; AFAC5(x_1, x_2, x_3, n) = (1, x_2, x_3, n) &gt; )</td>
</tr>
<tr>
<td>A4</td>
<td>( \forall(x_1, x_2, x_3) : &lt; AFAC4(x_1, x_2, x_3, n) = (x_1, 0, x_3, n) &gt; )</td>
</tr>
<tr>
<td>A3</td>
<td>( \forall(x_1, x_2, x_3) : &lt; AFAC3(x_1, x_2, x_3, n) = (x_1, x_2 + 1, x_3, n) &gt; )</td>
</tr>
<tr>
<td>A2</td>
<td>( \forall(x_1, x_2, x_3) : &lt; AFAC2(x_1, x_2, x_3, n) = (x_1 \cdot x_2, x_2, x_3, n) &gt; )</td>
</tr>
<tr>
<td>A1</td>
<td>( \forall(x_1, x_2, x_3) : &lt; AFAC1(x_1, x_2, x_3, n) = (x_1, x_2, x_1, n) &gt; )</td>
</tr>
<tr>
<td>G2</td>
<td>( \forall(x_1, x_2, x_3) : &lt; GFAC2(x_1, x_2, x_3, n) = AFAC4(AFAC5(x_1, x_2, x_3, n)) &gt; )</td>
</tr>
<tr>
<td>G1</td>
<td>( \forall(x_1, x_2, x_3) : &lt; GFAC1(x_1, x_2, x_3, n) = AFAC2(AFAC3(x_1, x_2, x_3, n)) &gt; )</td>
</tr>
<tr>
<td>W1a1</td>
<td>( \forall(x_1, x_2, x_3, x_4) : &lt; (x_4 = 0) &gt; )</td>
</tr>
<tr>
<td></td>
<td>( \rightarrow &lt; WFAC1A(x_1, x_2, x_3, n, x_4) = (x_1, x_2, x_3, n) &gt; )</td>
</tr>
<tr>
<td>W1a2</td>
<td>( \forall(x_1, x_2, x_3, x_4) : &lt; (0 &lt; x_4) &gt; )</td>
</tr>
<tr>
<td></td>
<td>( \rightarrow &lt; WFAC1A(x_1, x_2, x_3, n, x_4) = GFAC1(WFAC1A(x_1, x_2, x_3, n, x_4 - 1)) &gt; )</td>
</tr>
<tr>
<td>W1a3</td>
<td>( \forall(x_1, x_2, x_3, x_4) : \exists(v_1, v_2, v_3) : )</td>
</tr>
<tr>
<td></td>
<td>( &lt; (v_1, v_2, v_3, n) = WFAC1A(x_1, x_2, x_3, n, x_4) &gt; )</td>
</tr>
</tbody>
</table>

**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 \([D1]\):**

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

\( \rightarrow < (y_1, y_2, y_3, n) = (1, 0, x_3, n) > \)
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. [\(\rightarrow\) Intro 2.3-11] \(< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >\)
13. \(\rightarrow < (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >\)

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]:

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

\(\rightarrow < y_1 = fac(y_2) >\)

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]∀(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) >
   ←< (z_1, z_2, z_3, n) = (1, 0, w_3, n) >
4. [∀ Elim 3]< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >
   ←< (y_1, y_2, y_3, n) = (1, 0, x_3, n) >
5. [Intro 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. [Intro 2, 3-10]< (y_1, y_2, y_3, n) = GFAC2(x_1, x_2, x_3, n) >--< y_1 = fac(y_2) >
12.[∀ Intro 1, 2-11]∀(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) > ∧ < x_1 = fac(x_2) >)
   ←< 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] :

∀(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) > ∧ < x_1 = fac(x_2) >)

←< 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) > \land < x_1 = fac(x_2) >)
3. [\land Elim 2] < (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) >
4. [\land 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. [\exists 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 \cdot z_2 + 1 \cdot z_3, n) >
8. [\forall Elim 7] < AFAC3(x_1, x_2, x_3, n) = (x_1 \cdot x_2 + 1 \cdot x_3, n) >
9. [\equiv Sub 8, 6] < GFAC1(x_1, x_2, x_3, n) = AFAC2(x_1 \cdot x_2 + 1 \cdot x_3, n) >
10. [A2 RENAME] \forall (z_1, z_2, z_3) : < AFAC2(z_1, z_2, z_3, n) = (z_1 \cdot z_2 + z_2, z_3, n) >
11. [\forall Elim 10] < AFAC2(x_1, x_2 + 1, x_3, n) = (x_1 \cdot (x_2 + 1), x_2 + 1, x_3, n) >
12. [\equiv Sub 11, 9] < GFAC1(x_1, x_2, x_3, n) = (x_1 \cdot (x_2 + 1), x_2 + 1, x_3, n) >
13. [\equiv Sub 3, 12] < (y_1, y_2, y_3, n) = (x_1 \cdot (x_2 + 1), x_2 + 1, x_3, n) >
14. [\equiv Elim 13] < y_1 = x_1 \cdot (x_2 + 1) >
15. [\equiv Elim 13] < y_2 = x_2 + 1 >
16. [Number Theory] \forall (z_1, z_2) : < z_1 = fac(z_2) >
   \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \equiv < z_1 \cdot (z_2 + 1) = fac(z_2 + 1) >
17. [\forall Elim 16] < x_1 = fac(x_2) > < x_1 \cdot (x_2 + 1) = fac(x_2 + 1) >
18. [\equiv Elim 4, 17] < x_1 \cdot (x_2 + 1) = fac(x_2 + 1) >
19. [\equiv Sub 14, 18] < y_1 = fac(x_2 + 1) >
20. [\equiv Sub 15, 19] < y_1 = fac(y_2) >
21. [\equiv Intro 2, 3-20] < (y_1, y_2, y_3, n) = GFAC1(x_1, x_2, x_3, n) > < x_1 = fac(x_2) >
   \quad \quad \quad \quad \quad \equiv < 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) >
   \quad \quad \quad \quad \quad \land < x_1 = fac(x_2) > \equiv < 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) : < WFACE1A(x_1, x_2, x_3, n, 0) = (x_1, x_2, x_3, n) >
Proof 4:

1. **[W1a1 RENAME]** \(\forall(z_1, z_2, z_3, z_4) :< (z_4 = 0) >\)
   \[\rightarrow< WFACE1A(z_1, z_2, z_3, n, z_4) = (z_1, z_2, z_3, n) >\]
2. **[∀ Elim 1]** \(\forall(z_1, z_2, z_3) :< (0 = 0) >\)
   \[\rightarrow< WFACE1A(z_1, z_2, z_3, x_1, 0) = (z_1, z_2, z_3, n) >\]
3. **[Number Theory]** \(< 0 = 0 >\)
4. **[→ Elim 3.2]** \(\forall(z_1, z_2, z_3) :< WFACE1A(z_1, z_2, z_3, n, 0) = (z_1, z_2, z_3, n) >\)

Proof 5:

Proving [D5]:

\[\forall(y_1, y_2, y_3, x_1, x_2, x_3) : (< (y_1, y_2, y_3, n) = WFACE1A(x_1, x_2, x_3, n, 0) > \land < x_1 = fac(x_2) >) \]
\[\rightarrow< y_1 = fac(y_2) >\]

Proof 5:

1. **[Variables]** \((y_1, y_2, y_3, x_1, x_2, x_3)\)
2. **[Assumption]** \(< (y_1, y_2, y_3, n) = WFACE1A(x_1, x_2, x_3, n, 0) > \land < x_1 = fac(x_2) >\)
3. **[D4 RENAME]** \(\forall(z_1, z_2, z_3) :< WFACE1A(z_1, z_2, z_3, n, 0) = (z_1, z_2, z_3, n) >\)
4. **[∀ Elim 3]** \(< WFACE1A(x_1, x_2, x_3, n, 0) = (x_1, x_2, x_3, n) >\)
5. **[∧ Elim 2]** \(< (y_1, y_2, y_3, n) = WFACE1A(x_1, x_2, x_3, n, 0) >\)
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. **[∧ 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. **[→ Intro 2, 3-11]** \(< (y_1, y_2, y_3, n) = WFACE1A(x_1, x_2, x_3, n, 0) > \land < x_1 = fac(x_2) >\) \[\rightarrow< y_1 = fac(y_2) >\]
13. **[∀ Intro 1, 2-12]** \(\forall(y_1, y_2, y_3, x_1, x_2, x_3) : (< (y_1, y_2, y_3, n) = WFACE1A(x_1, x_2, x_3, n, 0) > \land < x_1 = fac(x_2) >) \rightarrow< y_1 = fac(y_2) >\)
Proof 6:

Proof 5 was a proof of the base case of the factorial property that we are trying to demonstrate is satisfied by \( W\text{FAC1} \). This is the general case. Proving [D6]:

\[
(\forall (y_1, y_2, y_3, x_1, x_2, x_3) : \\
(\langle y_1, y_2, y_3, n \rangle = W\text{FAC1}A(x_1, x_2, x_3, n, ind - 1) \land \langle x_1 = \text{fac}(x_2) \rangle \\
\to \langle y_1 = \text{fac}(y_2) \rangle) \\
\to \\
(\forall (y_1, y_2, y_3, x_1, x_2, x_3) : \\
(\langle y_1, y_2, y_3, n \rangle = W\text{FAC1}A(x_1, x_2, x_3, n, ind) \land \langle x_1 = \text{fac}(x_2) \rangle \\
\to \langle y_1 = \text{fac}(y_2) \rangle)\
\]

Proof 6:

1. [Assumption] ∀(y₁, y₂, y₃, x₁, x₂, x₃) :
   
   
   
   ( < (y₁, y₂, y₃, n) = WFACE₁A(x₁, x₂, x₃, n, ind - 1) > ∧ < x₁ = fac(x₂) > ) → < y₁ = fac(y₂) >

2. [Variables] (z₁, z₂, z₃, w₁, w₂, w₃)

3. [Assumption] ( < (z₁, z₂, z₃, n) = WFACE₁A(w₁, w₂, w₃, n, ind) > ∧ < w₁ = fac(w₂) >)

4. [Number Theory] (∀x : ( < 0 < x > → ( < 0 = x - 1 > ∨ < 0 < x - 1 > )))

5. [∀ Elim 4] ( < 0 < ind > → ( < 0 = ind - 1 > ∨ < 0 < ind - 1 > ))

6. [Induction 1] < 0 < ind >

7. [→ Elim 6, 5] < 0 = ind - 1 > ∨ < 0 < ind - 1 >

8. [And Elim 3] ( < z₁, z₂, z₃, n ) = WFACE₁A(w₁, w₂, w₃, n, ind) >

9. [W₁a²] ∀(y₁, y₂, y₃, y₄) :
   
   
   
   ( < 0 < y₄ > → WFACE₁A(y₁, y₂, y₃, n, y₄) = GFAC₁(WFACE₁A(y₁, y₂, y₃, n, y₄ - 1) >)

10. [∀ Elim 9] ( < 0 < ind > →

    < WFACE₁A(w₁, w₂, w₃, n, ind) = GFAC₁(WFACE₁A(w₁, w₂, w₃, n, ind - 1) >)

11. [→ Elim 6, 10]
    
    ( < WFACE₁A(w₁, w₂, w₃, n, ind) = GFAC₁(WFACE₁A(w₁, w₂, w₃, n, ind - 1) >)

12. [Assumption] < 0 = ind - 1 >

13. [= Elim 11, 12] < WFACE₁A(w₁, w₂, w₃, n, ind) = GFAC₁(WFACE₁A(w₁, w₂, w₃, n, 0) >

14. [Dₜ] ∀(y₁, y₂, y₃) : WFACE₁A(y₁, y₂, y₃, n, 0) = (y₁, y₂, y₃, n) >

15. [∀ Elim 14] < WFACE₁A(w₁, w₂, w₃, n, 0) = (w₁, w₂, w₃, n) >

16. [= Elim 15, 13] < WFACE₁A(w₁, w₂, w₃, n, ind) = GFAC₁(w₁, w₂, w₃, n) >

17. [And Elim 3] (z₁, z₂, z₃, n ) = WFACE₁A(w₁, w₂, w₃, n, ind) >

18. [= Elim 17, 16] ( < z₁, z₂, z₃, n ) = GFAC₁(w₁, w₂, w₃, n) >

19. [And Elim 3] < w₁ = fac(w₂) >

20. [And Intro 18, 19] ( < z₁, z₂, z₃, n ) = GFAC₁(w₁, w₂, w₃, n) > ∧ < w₁ = fac(w₂) >

21. [Dₜ] ∀(y₁, y₂, y₃, x₁, x₂, x₃) :
    
    
    
    ( < y₁, y₂, y₃, n ) = GFAC₁(x₁, x₂, x₃, n) > ∧ < x₁ = fac(x₂) > ) → < y₁ = fac(y₂) >

22. [∀ Elim 21] ( < z₁, z₂, z₃, n ) = GFAC₁(w₁, w₂, w₃, n) > ∧ < w₁ = fac(w₂) >

       → < z₁ = fac(z₃) >
\[ \text{[→ Elim 20, 22]} < z_1 = \text{fac}(z_2) > \]

501. \[\text{[Assumption]} < 0 < \text{ind} - 1 >\]

502. \[\text{[W1a3]} \forall (x_1, x_2, x_3, x_4) : \exists (v_1, v_2, v_3, n) = \text{WFC1A}(x_1, x_2, x_3, n, x_4)\]

503. \[\text{[∀ Elim 502]} \exists (v_1, v_2, v_3, n) = \text{WFC1A}(x_1, x_2, x_3, n, \text{ind} - 1)\]

504. \[\text{[Variables]} (v_1^0, v_2^0, v_3^0) \text{ (v}_1^0, v_2^0, v_3^0, n) = \text{WFC1A}(x_1, x_2, x_3, n, \text{ind} - 1)\]

505. \[\forall \text{ Elim 1} < (v_1^0, v_2^0, v_3^0, n) = \text{WFC1A}(w_1, w_2, w_3, n, \text{ind} - 1) > \]
\[\land < w_1 = \text{fac}(w_2) > \longrightarrow < v_1^0 = \text{fac}(v_2^0) >\]

506. \[\text{[And Elim 3]} < w_1 = \text{fac}(w_2) >\]

507. \[\text{[And Intro 504, 506]} < (v_1^0, v_2^0, v_3^0, n) = \text{WFC1A}(w_1, w_2, w_3, n, \text{ind} - 1) > \]
\[\land < w_1 = \text{fac}(w_2) >\]

508. \[\text{[→ Elim 507, 505]} < v_1^0 = \text{fac}(v_2^0) >\]

509. \[\text{[=Sub 504, 11]} < \text{WFC1A}(w_1, w_2, w_3, n, \text{ind}) = \text{GFAC1}(v_1^0, v_2^0, v_3^0, n) >\]

510. \[\text{[=Sub 8, 509]} < (z_1, z_2, z_3, n) = \text{GFAC1}(v_1^0, v_2^0, v_3^0, n) >\]

511. \[\text{[D3]} \forall (y_1, y_2, y_3, x_1, x_2, x_3) : \]
\[< (y_1, y_2, y_3, n) = \text{GFAC1}(x_1, x_2, x_3, n) > \land < x_1 = \text{fac}(x_2) > \]
\[\longrightarrow < y_1 = \text{fac}(y_2) >\]

512. \[\forall \text{ Elim 511} < (z_1, z_2, z_3, n) = \text{GFAC1}(v_1^0, v_2^0, v_3^0, n) > \land < v_1^0 = \text{fac}(v_2^0) > \]
\[\longrightarrow < z_1 = \text{fac}(z_2) >\]

513. \[\land \text{ Intro 510, 508} < (z_1, z_2, z_3, n) = \text{GFAC1}(v_1^0, v_2^0, v_3^0, n) > \land < v_1^0 = \text{fac}(v_2^0) >\]

994. \[\text{[→ Elim 512, 513]} < z_1 = \text{fac}(z_2) >\]

995. \[\exists \text{ Elim 503, 504-994} < z_1 = \text{fac}(z_2) >\]

996. \[\forall \text{ Elim 7, 12-500. 501-995} < z_1 = \text{fac}(z_2) >\]

997. \[\text{[→ Intro 3, 4-996]}< (z_1, z_2, z_3, n) = \text{WFC1A}(w_1, w_2, w_3, n, \text{ind}) > \land < w_1 = \text{fac}(w_2) > \]
\[\longrightarrow < z_1 = \text{fac}(z_2) >\]

998. \[\forall \text{ Intro 2-997} \forall (z_1, z_2, z_3, w_1, w_2, w_3) : \]
\[< (z_1, z_2, z_3, n) = \text{WFC1A}(w_1, w_2, w_3, n, \text{ind}) > \land < w_1 = \text{fac}(w_2) > \longrightarrow < z_1 = \text{fac}(z_2) >\]

999. \[\text{[998 Rename]} \forall (y_1, y_2, y_3, x_1, x_2, x_3) : \]
\[< (y_1, y_2, y_3, n) = \text{WFC1A}(x_1, x_2, x_3, n, \text{ind}) > \land < x_1 = \text{fac}(x_2) > \longrightarrow < y_1 = \text{fac}(y_2) >\]

1000. \[\forall (y_1, y_2, y_3, x_1, x_2, x_3) : \]
\[< (y_1, y_2, y_3, n) = \text{WFC1A}(x_1, x_2, x_3, n, \text{ind} - 1) > \land < x_1 = \text{fac}(x_2) > \longrightarrow < y_1 = \text{fac}(y_2) >\]

The other $S_L$ calculus rules are now introduced.

- Collapse rule:

\[ \text{56} \]
The collapse rule allows us to derive something of the form:

\[
\text{FUNC1}(\text{FOO}(x, y), n) = \text{FUNC1}(a, b, n) \text{ from } \text{FOO}(x, y) = (a, b).
\]

If we didn’t have such a rule, we would be stuck at \(\text{FUNC1}(\text{FOO}(x, y), n) = ((a, b), n)\) with no way to “collapse” the inner parenthesis.

- Induction rules

\[
\underbrace{< 0 < \text{ind} >} \ \text{ind}
\]

\[
\frac{F\{x \Rightarrow 0\} \quad F\{x \Rightarrow (\text{ind} - 1)\}}{\forall (x): F} \quad \text{Induction}
\]

**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]:

\[
\forall (w, y_1, y_2, y_3, x_1, x_2, x_3):
\]

\[
(< (y_1, y_2, y_3, n) = \text{WFACT}(x_1, x_2, x_3, n, w) > \land < x_1 = \text{fac}(x_2) >) \\
\quad \quad \rightarrow (< y_1 = \text{fac}(y_2) >)
\]
Proof 7:

1. [D5] \[ \forall(y_1, y_2, y_3, x_1, x_2, x_3) : \\
\quad (\langle y_1, y_2, y_3, n \rangle = W\text{FA}C1A(x_1, x_2, x_3, n, 0) > \land x_1 = \text{fac}(x_2)) \rightarrow (\langle y_1 = \text{fac}(y_2) >) \]

2. [D6] \[ \forall(y_1, y_2, y_3, x_1, x_2, x_3) : \\
\quad (\langle y_1, y_2, y_3, n \rangle = W\text{FA}C1A(x_1, x_2, x_3, n, \text{ind} - 1) > \land x_1 = \text{fac}(x_2)) \rightarrow (\langle y_1 = \text{fac}(y_2) >) \]

3. [Induction 1, 2] \[ \forall(w) : \forall(y_1, y_2, y_3, x_1, x_2, x_3) : \\
\quad (\langle y_1, y_2, y_3, n \rangle = W\text{FA}C1A(x_1, x_2, x_3, n, w) > \land x_1 = \text{fac}(x_2)) \rightarrow (\langle y_1 = \text{fac}(y_2) >) \]

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. \( \Lambda_M \) is a finite set of variables

2. \( \Gamma_M \) is a finite set of well-formed \( S_L \) expressions such that for any \( \gamma \in \Gamma_M \), if the variable \( x \) occurs in \( \gamma \), then either \( x \) is bound in \( \gamma \) or \( x \in \Lambda_M \)

3. \( \Upsilon_M \) 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 \( \mathbb{N} \) 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. \( \forall(x,y) : <(x + 1) = (y + 1) > \rightarrow <x = y> \)
3. \( \forall(x) : (<x = 0>) \lor (\exists(y) : <(y + 1) = x>) \)
4. \( \forall(x) : <x + 0 = x> \)
5. \( \forall(x,y) : <x + (y + 1) = (x + y) + 1> \)
6. \( \forall(x) : <x \cdot 0 = 0> \)
7. \( \forall(x,y) : <x \cdot (y + 1) = (x \cdot y) + x> \)
8. \( \forall(x) : <\exp(x,0) = 1> \)
9. \( \forall(x,y) : <\exp(x,(y + 1)) = \exp(x,y) \cdot x> \)
10. \( \forall(x) : <x < x + 1> \)
11. \( \forall(x,y) : (<x < y>) \rightarrow ((<x + 1 < y>) \lor (<x + 1 = y>)) \)
12. \( \forall(x,y) : \neg(<x < y>) \rightarrow ((<x = y>) \lor (<y < x>)) \)
13. \( \forall(x,y) : ((<x = y>) \lor (y < x)) \rightarrow \neg(<x < y>) \)
14. \( \forall(x,y,z) : ((<x < y>) \land (<y < z>)) \rightarrow (<x < z>) \)

We also add to the set:

1. \( <\text{fac}(0) = 1> \)
2. \( \forall (x) \ldotp \langle fac(x + 1) = fac(x) \cdot (x + 1) \rangle \)

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 $\Gamma_{M_P}$ describes the classes of functions (Tuple-Property classes) that the statements of $P$ define. To build $M_P$, $SMP$ adds sentences to $\Gamma_{M_P}$ and variables to $\Lambda_{M_P}$. $\Gamma_{M_P}$ is an empty set when the algorithm starts, while $\Lambda_{M_P}$ contains $P$'s parameters which are used as constants. Since $P$ is itself a $C^c$ statement (see chapter 3), $SMP$ also adds to $\Gamma_{M_P}$ 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 $f : \mathbb{N}^p \Longrightarrow \mathbb{N}^p$ 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,

\[
[(y_1, \ldots, y_p) = f(v_1, \ldots, v_p)] \quad \text{PRE CONDITION}
\]

\[
\text{FUNC}
\]

\[
[(y_1 = v_1) \land \ldots \land (y_p = v_p)] \quad \text{POST CONDITION}
\]

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 $f : \mathbb{N}^n \Longrightarrow \mathbb{N}^m$ can be encoded as a function in $\mathbb{N}^p \Longrightarrow \mathbb{N}^p$ where $p \geq n, m$.

**Remark 5.1.4** We notice that every recursive function from $\mathbb{N}^p$ to $\mathbb{N}^p$ 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:

\[(y_1 = v_1) \land \ldots \land (y_p = v_p)] \quad \text{PRE CONDITION}\]

\[\text{FUNC}\]

\[(v_1, \ldots, v_p) = f(y_1, \ldots, y_p)] \quad \text{POST CONDITION}\]

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

\[
\begin{align*}
(y_1 = v_1) \land \ldots \land (y_p = v_p) & \land ((y_1^0, \ldots, y_p^0) = f(v_1, \ldots, v_p)) \quad \text{PRE CONDITION} \\
(y_1 = v_1) \land \ldots \land (y_p = v_p) & \land ((y_1^0, \ldots, y_p^0) = f(y_1, \ldots, y_p)) \quad \text{IMPLIED}
\end{align*}
\]

\[\text{FUNC}\]

\[
\begin{align*}
(y_1^0 = v_1) \land \ldots \land (y_p^0 = v_p) & \quad \text{(from code equivalence of } f \text{ and PRE CONDITION)} \quad \text{IMPLIED} \\
(y_1^0, \ldots, y_p^0) & = f(y_1, \ldots, y_p) \quad \text{IMPLIED} \\
(v_1, \ldots, v_p) & = f(y_1, \ldots, y_p) \quad \text{IMPLIED}
\end{align*}
\]

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 \text{ is from } \mathbb{N}^p \text{ 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 \rightarrow \mathbb{N}^p are two functions Code-Equivalent to FUNC1 and to FUNC2 respectively, then the function defined by composition as:

\[f_{1,2} = f_2 \circ f_1 : \mathbb{N}^p \rightarrow \mathbb{N}^p\]

is Code-Equivalent to the statement \([\text{FUNC}] = [\text{FUNC1}][\text{FUNC2}]\), the statement constructed from FUNC1 followed immediately by FUNC2.

Proof. P is in the form:
Let \((y_1, \ldots, y_p), (y^1_1, \ldots, y^2_1), (y^2_1, \ldots, y^2_p)\) be 3 tuples of logical variables that do not appear anywhere in \(P\) and chosen such that \(|(y_1, \ldots, y_p) = f_{1,2}(v_1, \ldots, v_p)|, |(y^1_1, \ldots, y^2_1) = f_1(v_1, \ldots, v_p)|\) and \(|(y^2_1, \ldots, y^2_p) = f_2(v_1, \ldots, v_p)|\) hold respectively just before the execution of \(FUNC\). \(FUNC1\) and \(FUNC2\).

We then construct the following Hoare Triple proof:

\[
\begin{array}{c|l}
|\!(y_1, \ldots, y_p) = f_{1,2}(v_1, \ldots, v_p) \land (y^1_1, \ldots, y^1_p) = f_1(v_1, \ldots, v_p)| & PRE\ CONDITION \\
|\!(y_1, \ldots, y_p) = f_2 \circ f_1(v_1, \ldots, v_p) \land (y^1_1, \ldots, y^1_p) = f_1(v_1, \ldots, v_p)| & IMPLIED \\
|\!(y_1, \ldots, y_p) = f_1(v_1, \ldots, v_p) \land (y^1_1, \ldots, y^1_p) = f_1(v_1, \ldots, v_p)| & IMPLIED \\
\end{array}
\]

\(FUNC1\)
\begin{center}
\begin{tabular}{|l|l|}
\hline
\((y_1, \ldots, y_p) = f_2(y_1^1, \ldots, y_p^1)\
\land (y_1^1 = v_1) \land \ldots \land (y_p^1 = v_p)\) & \(\text{(} f_1 \text{ Code-Equivalent to FUNC1)}\) \\
\hline
\((y_1, \ldots, y_p) = f_2(v_1, \ldots, v_p)\) & \text{IMPLIED} \\
\hline
\((y_1, \ldots, y_p) = f_2(v_1, \ldots, v_p)\) & \text{IMPLIED} \\
\hline
\((y_1^2, \ldots, y_p^2) = f_2(v_1, \ldots, v_p)\) & \text{IMPLIED} \\
\hline
\end{tabular}
\end{center}

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 \(\mathbb{N}^p\) to \(\mathbb{N}\). Then, for every \((c_1, \ldots, c_p) \in \mathbb{N}^p\), if \(h(f(c_1, \ldots, c_p)) = a\), there is a Hoare Triple derivation of:

\[
\begin{align*}
\{(v_1 = c_1) \land \ldots \land (v_p = c_p)\} & \quad \text{PRE CONDITION} \\
\hline
\text{FUNC} \\
\hline
|h(v_1, \ldots, v_p) = a| & \quad \text{POST CONDITION}
\end{align*}
\]

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 \( \text{FUNC} \), the final state of \( \text{FUNC} \)'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 \( \text{SMP} \) algorithm. This comes from the Code-Equivalency properties of the functions that are defined through \( \text{SMP} \).

**Proof.**

Since \( f \) is code-equivalent to \( \text{FUNC} \), there is an Hoare Triple proof of:

\[
\begin{align*}
(y_1, \ldots, y_p) &= f(v_1, \ldots, v_p) & \text{PRE CONDITION} \\
\text{FUNC} \\
(y_1 = v_1) \land \ldots \land (y_p = v_p) & \text{POST CONDITION}
\end{align*}
\]

Since \( f \) is from \( \text{NP} \) to \( \text{NP} \) 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 \( \text{NP} \) to \( \text{N} \). Therefore, there is a Hoare Triple derivation of:

\[
\begin{align*}
(y_1, \ldots, y_p) &= f(v_1, \ldots, v_p) \land (v_1 = c_1) \land \ldots \land (v_p = c_p) & \text{PRE CONDITION} \\
(y_1 = f_1(v_1, \ldots, v_p)) \land \ldots \land (y_p = f_p(v_1, \ldots, v_p)) \\
(y_1, \ldots, y_p) &= f(v_1, \ldots, v_p) \land (v_1 = c_1) \land \ldots \land (v_p = c_p) & \text{IMPLIED} \\
(y_1 = f_1(c_1, \ldots, c_p)) \land \ldots \land (y_p = f_p(c_1, \ldots, c_p)) \\
(y_1, \ldots, y_p) &= f(v_1, \ldots, v_p) & \text{IMPLIED}
\end{align*}
\]

\( \text{FUNC} \)
\[(y_1 = f_1(c_1, \ldots, c_p)) \land \ldots \land (y_P = f_P(c_1, \ldots, c_p)) \land (y_1 = v_1) \land \ldots \land (y_P = v_P)\] (By Code-Equivalence of f)

\[|(v_1 = f_1(c_1, \ldots, c_p)) \land \ldots \land (v_P = f_P(c_1, \ldots, c_p))|\] 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 MP which it modifies during its execution. Initially, $\Gamma_{MP}$ is empty and $\Lambda_{MP}$ 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 $\Gamma_{MP}$. 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

67
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(ASj)

$\Gamma_M ::= \Gamma_M \cup$

$\{ \forall (v_1, \ldots , v_p) :$

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

$v_{i+1}, \ldots , v_p, n_1, \ldots , n_q)\}$

$\Gamma_M ::= \Gamma_M \cup$

$\{ \forall (v_1, \ldots , v_p) :$

$\exists (y_1, \ldots , y_p) :$

$(y_1, \ldots , y_p, n_1, \ldots , n_q) = A[P][j](v_1, \ldots , v_p, n_1, \ldots , n_q)\}$

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]$x_1 = (n_1 + 1) \times 5;$
[3]$x_2 = 3 \times (n_2 + n_1);$  
[2]$x_3 = x_1 + x_2;$  
[1]$x_4 = x_2 + x_3;$
}

69
model : Poly

Constants : $x_1, n_2$

$\forall(x_1, x_2, x_3, x_4) : (Poly(x_1, x_2, x_3, x_4, n_1, n_2) = (APoly1(APoly2(APoly3(APoly4(x_1, x_2, x_3, x_4, n_1, n_2))))))$

$\Gamma_M :$

Defining : $APoly1$ from assignment statement $\neg[1]x_4 = x_2 + x_3;-$

$\forall(x_1, x_2, x_3, x_4) : (APoly1(x_1, x_2, x_3, x_4, n_1, n_2) = (x_1, x_2, x_3, (x_2 + x_3), n_1, n_2))$

$\forall(x_1, x_2, x_3, x_4) : (\exists(y_1, y_2, y_3, y_4) : ((y_1, y_2, y_3, y_4) = APoly1(x_1, x_2, x_3, x_4, n_1, n_2)))$

End Definition ($APoly1$)

Defining : $APoly2$ from assignment statement $\neg[2]x_3 = x_1 + x_2;-$

$\forall(x_1, x_2, x_3, x_4) : (APoly2(x_1, x_2, x_3, x_4, n_1, n_2) = (x_1, x_2, (x_1 + x_2), x_4, n_1, n_2))$

$\forall(x_1, x_2, x_3, x_4) : (\exists(y_1, y_2, y_3, y_4) : ((y_1, y_2, y_3, y_4) = APoly2(x_1, x_2, x_3, x_4, n_1, n_2)))$

End Definition ($APoly2$)

Defining : $APoly3$ from assignment statement $\neg[3]x_2 = 3 \cdot (n_2 + n_1);$-

$\forall(x_1, x_2, x_3, x_4) : (APoly3(x_1, x_2, x_3, x_4, n_1, n_2) = (x_1, (3 \cdot (n_2 + n_1)), x_3, x_4, n_1, n_2))$

$\forall(x_1, x_2, x_3, x_4) : (\exists(y_1, y_2, y_3, y_4) : ((y_1, y_2, y_3, y_4) = APoly3(x_1, x_2, x_3, x_4, n_1, n_2)))$

End Definition ($APoly3$)

Defining : $APoly4$ from assignment statement $\neg[4]x_1 = (n_1 + 1) \cdot 5;-$

$\forall(x_1, x_2, x_3, x_4) : (APoly4(x_1, x_2, x_3, x_4, n_1, n_2) = (((n_1 + 1) \cdot 5), x_2, x_3, x_4, n_1, n_2))$

$\forall(x_1, x_2, x_3, x_4) : (\exists(y_1, y_2, y_3, y_4) : ((y_1, y_2, y_3, y_4) = APoly4(x_1, x_2, x_3, x_4, n_1, n_2)))$

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

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

holds just before the execution of the statement $S$.  

70
If $S$ is an assignment statement, $P$ is in the form:

$$P(n_1, \ldots, n_q)$$

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

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**

$$[(y_1 = v_1) \land \ldots \land (y_i = F_j(v_1, \ldots, v_p, n_1, \ldots, n_p)) \land \ldots (y_p = v_p)]$$

**IMPLIED**

(This follows from the definition of $AP_j$)

$$[j] \ v_[i] = F_j(v_1, \ldots, v_p, n_1, \ldots, n_q);$$

$$[(y_1 = v_1) \land \ldots \land (y_i = v_i) \land \ldots (y_p = v_p)]$$

**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:

\[
\text{If(COND)}
\begin{cases}
\quad \text{[If Part]} \\
\text{Else} \\
\quad \text{[Else Part]}
\end{cases}
\]

BEGIN ProcessIf(\(IfStatement_j\))

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

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

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

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

Return \(I[P][j]\)

END ProcessIf

Remark 5.1.11 ProcessCond(\(Conditionalexpression\)) simply replaces the \('=='\) \(C^c\) operator by the \(S_L = \text{symbol}\) (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]$x_1 = n_1$;
    }
  }
}
<table>
<thead>
<tr>
<th>model : max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants : ( n_1 n_2 n_3 )</td>
</tr>
<tr>
<td>( \forall(x_1) : (\text{max}(x_1, n_1, n_2, n_3) = (\text{Imax3}(x_1, n_1, n_2, n_3))) )</td>
</tr>
</tbody>
</table>

\[ \Gamma_M : \]

Defining : \( \text{Amax4} \) from assignment statement :\([-4]x_1 = n_3:\) -

\[ \forall(x_1) : (\text{Amax4}(x_1, n_1, n_2, n_3) = ((n_3), n_1, n_2, n_3)) \]
\[ \forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = \text{Amax4}(x_1, n_1, n_2, n_3))) \]

End Definition (\( \text{Amax4} \))

Defining : \( \text{Amax3} \) from assignment statement :\([-3]x_1 = n_2:\) -

\[ \forall(x_1) : (\text{Amax3}(x_1, n_1, n_2, n_3) = ((n_2), n_1, n_2, n_3)) \]
\[ \forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = \text{Amax3}(x_1, n_1, n_2, n_3))) \]

End Definition (\( \text{Amax3} \))

Defining : \( \text{Imax1} \) from if statement :\( \text{if } (n_2 < n_3) \{ \ldots \} \text{ else } \{ \ldots \} -\)

\[ \forall(x_1) : ((n_2 < n_3) \rightarrow (\text{Imax1}(x_1, n_1, n_2, n_3) = (\text{Amax4}(x_1, n_1, n_2, n_3)))) \]
\[ \forall(x_1) : ((\neg(n_2 < n_3)) \rightarrow (\text{Imax1}(x_1, n_1, n_2, n_3) = (\text{Amax3}(x_1, n_1, n_2, n_3)))) \]
\[ \forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = \text{Imax1}(x_1, n_1, n_2, n_3))) \]

End Definition (\( \text{Imax1} \))

Defining : \( \text{Amax2} \) from assignment statement :\([-2]x_1 = n_3:\) -

\[ \forall(x_1) : (\text{Amax2}(x_1, n_1, n_2, n_3) = ((n_3), n_1, n_2, n_3)) \]
\[ \forall(x_1) : (\exists(y_1) : ((y_1, n_1, n_2, n_3) = \text{Amax2}(x_1, n_1, n_2, n_3))) \]

End Definition (\( \text{Amax2} \))
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

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

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

```plaintext
P(n_1, \ldots, n_q)
{
  v_1 = \ldots;
  v_2 = \ldots;
  \ldots
```
\[ v_p = \ldots; \]
\[
\text{If}(\text{COND}) \quad // (S)
\{
  \text{[If Part]}
\}
\text{Else}
\{
  \text{[Else Part]}
\}
\]

Where we assume that the functions \( I_f_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \) and \( E_l_s_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \) defined by the SMP algorithm on the [If Part] and the [Else Part] statements satisfy Code-Equivalency.

\( IP_j \) is in the general form:

\[
(\text{Cond}) \longrightarrow IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) = I_f_j(v_1, \ldots, v_p, n_1, \ldots, n_q)
\]

\[
(\neg \text{Cond}) \longrightarrow IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) = E_l_s_j(v_1, \ldots, v_p, n_1, \ldots, n_q)
\]

We then derive the following Hoare Triple Proof:

\[
[(y_1, \ldots, y_p, n_1, \ldots, n_q) = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q)] \mid \text{PRE CONDITION}
\]

\text{If}(\text{COND}) \{
| (Cond) \[ y_1, \ldots, y_p, n_1, \ldots, n_q \] = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \] | IF STATEMENT |

| (Cond) \[ y_1, \ldots, y_p, n_1, \ldots, n_q \] = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \] | IMPLIED |

(Follows from definition of IP_j, see above)

| (IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) = If_j(v_1, \ldots, n_q)) \] \[ y_1, \ldots, y_p, n_1, \ldots, n_q \] = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \] | IMPLIED |

| ((y_1, \ldots, y_p, n_1, \ldots, n_q) = If_j(v_1, \ldots, v_p, n_1, \ldots, n_q)) \] | IMPLIED |

[If Part]

\[ (y_1 = v_1) \land \ldots \land (y_i = v_i) \land \ldots (y_p = v_p) \] (By Induction Hypothesis)

} Else {

| !-(Cond) \[ y_1, \ldots, y_p, n_1, \ldots, n_q \] = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \] | IF STATEMENT |

| !-(Cond) \[ y_1, \ldots, y_p, n_1, \ldots, n_q \] = IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) \] | IMPLIED |

\[ (IP_j(v_1, \ldots, v_p, n_1, \ldots, n_q) = Else_j(v_1, \ldots, v_p, n_1, \ldots, n_q)) \] \[ y_1, \ldots, y_p, n_1, \ldots, n_q \] = 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, \ldots, v_p, n_1, \ldots, n_q) \] | IMPLIED |

[Else Part]

\[ y_1 = v_1 \land \ldots \land y_i = v_i \land \ldots \land y_p = v_p \] (By Induction Hypothesis)
\[ y_1 = v_1 \land \ldots \land y_i = v_i \land \ldots y_p = v_p \mid IF \ \text{STATEMENT} \]

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

□
5.1.5 ProcessWhile

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

```plaintext
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) \rightarrow (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) \rightarrow (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:

$\text{Vars : } x_1 \ x_2 \ x_3$

$\text{Constants : } n_1 \ n_2$
```plaintext
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;
    }
}
```

<table>
<thead>
<tr>
<th>Model</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>n_1, n_2</td>
</tr>
<tr>
<td>∀(x_1, x_2, x_3) : (func(x_1, x_2, x_3, n_1, n_2) = (Wfunc(func4(func5(func6(x_1, x_2, x_3, n_1, n_2))))))</td>
<td></td>
</tr>
</tbody>
</table>

Γ_M: Defining : Afunc1 from assignment statement :-[1] x_3 = x_3 * x_1:-

∀(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))

∀(x_1, x_2, x_3) : (∃(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:-

∀(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))

∀(x_1, x_2, x_3) : (∃(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:-

∀(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))

∀(x_1, x_2, x_3) : (∃(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: \( W_{\text{func}1} \) from while statement \( \text{:-while}(x_1 < n_2)\{\ldots\} \)

\[ \forall(x_1, x_2, x_3, w) : ((0 = w) \rightarrow (W_{\text{func}1}(x_1, x_2, x_3, n_1, n_2, w) = (x_1, x_2, x_3, n_1, n_2))) \]
\[ \forall(x_1, x_2, x_3, w) : \\
((0 < w) \\
\rightarrow (W_{\text{func}1}(x_1, x_2, x_3, n_1, n_2, w) = A_{\text{func}1}(A_{\text{func}2}(A_{\text{func}3}(W_{\text{func}1}(x_1, x_2, x_3, n_1, n_2, (w - 1))))))) \]
\[ \forall(x_1, x_2, x_3, w) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3) = W_{\text{func}1}(x_1, x_2, x_3, n_1, n_2, w))) \]
End Definition (\( W_{\text{func}1} \))

Defining: \( A_{\text{func}4} \) from assignment statement \( \text{-[4]} x_3 = 1 \):

\[ \forall(x_1, x_2, x_3) : (A_{\text{func}4}(x_1, x_2, x_3, n_1, n_2) = (x_1, x_2, (1), n_1, n_2)) \]
\[ \forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = A_{\text{func}4}(x_1, x_2, x_3, n_1, n_2))) \]
End Definition (\( A_{\text{func}4} \))

Defining: \( A_{\text{func}5} \) from assignment statement \( \text{-[5]} x_2 = 1 \):

\[ \forall(x_1, x_2, x_3) : (A_{\text{func}5}(x_1, x_2, x_3, n_1, n_2) = (x_1, (1), x_3, n_1, n_2)) \]
\[ \forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = A_{\text{func}5}(x_1, x_2, x_3, n_1, n_2))) \]
End Definition (\( A_{\text{func}5} \))

Defining: \( A_{\text{func}6} \) from assignment statement \( \text{-[6]} x_1 = 0 \):

\[ \forall(x_1, x_2, x_3) : (A_{\text{func}6}(x_1, x_2, x_3, n_1, n_2) = ((0), x_2, x_3, n_1, n_2)) \]
\[ \forall(x_1, x_2, x_3) : (\exists(y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = A_{\text{func}6}(x_1, x_2, x_3, n_1, n_2))) \]
End Definition (\( A_{\text{func}6} \))

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:

\[
\text{While}(x_1, \ldots, x_p)\
\text{if Not (Cond) Evaluate to (x_1, \ldots, x_p)}
\]
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:

\[
LLOOP_1(x_1, \ldots, x_p, 0) = (x_1, \ldots, x_p)
\]

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 \(LOOP_1\) as:

\[
LLOOP_1(x_1, \ldots, x_p, n) = LOOP(LLOOP_1(x_1, \ldots, x_p, n - 1))
\]

Given a well formed definition of \(LOOP\):

82
• \(LLOOP1(x_1, \ldots, x_p, 0) = (x_1, \ldots, x_p)\)

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

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:

\[
P(n_1, \ldots, n_q)\]

\[
\{
  v_1 = \ldots;
  v_2 = \ldots;
  \ldots
  v_p = \ldots;
  \ldots
  \text{while(COND) } // (S)
  \{
  \quad \text{[While Body Part]}
  \}
\]

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 \(\mathbb{N}^{p+q}\) to \(\mathbb{N}^{p+q}\) generated by \(WP_j\) and defined as:

\[
FW = \{ f_k | f_k(v_1, \ldots, v_p, n_1, \ldots, n_q) = WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, k), k \in \mathbb{N} \}\}
\]
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:

  $$(y_1, \ldots, y_p, n_1, \ldots, n_q) = f_0(v_1, \ldots, v_p, n_1, \ldots, n_q)$$  
  $$(y_1, \ldots, y_p, n_1, \ldots, n_q) = WP_2(v_1, \ldots, v_p, n_1, \ldots, n_q, 0)$$  
  $$(y_1, \ldots, y_p, n_1, \ldots, n_q) = (v_1, \ldots, v_p, n_1, \ldots, n_q)$$

  **PRE CONDITION**

  **IMPLIED**

  **IMPLIED**

  while(COND) {
  
  [This doesn’t execute]
  
  }

  $$y_1 = v_1 \land \ldots \land y_i = v_i \land \ldots \land y_p = v_p$$  
  **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:

\[
\begin{align*}
(y_1, \ldots, y_p, n_1, \ldots, n_q) &= f_{t+1}(v_1, \ldots, v_p, n_1, \ldots, n_q) & \text{PRE CONDITION} \\
(y_1, \ldots, y_p, n_1, \ldots, n_q) &= WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, t + 1) & \text{IMPLIED} \\
(y_1, \ldots, y_p, n_1, \ldots, n_q) &= Body_j(WP_j(v_1, \ldots, v_p, n_1, \ldots, n_q, t)) & \text{IMPLIED} \\
(y_1, \ldots, y_p, n_1, \ldots, n_q) &= Body_j(f_t(v_1, \ldots, v_p, n_1, \ldots, n_q)) & \text{IMPLIED}
\end{align*}
\]

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:

\[
\begin{align*}
(y_1, \ldots, y_p, n_1, \ldots, n_q) &= f_{t+1}(v_1, \ldots, v_p, n_1, \ldots, n_q) & \text{PRE CONDITION} \\
& \vdots & \\
(y_1, \ldots, y_p, n_1, \ldots, n_q) &= Body_j(f_t(v_1, \ldots, v_p, n_1, \ldots, n_q)) & \text{IMPLIED}
\end{align*}
\]

\[
\begin{array}{l}
\text{while(cond)} \\\n\{ \\\n\text{//This executes t+1 times} \\\n\} \\
\begin{align*}
y_1 &= v_1 \land \ldots \land y_i = v_i \land \ldots \land y_p = v_p & (\text{By Code Equivalence of Body}_j \circ f_t) \\
\end{align*}
\end{array}
\]

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 $(v_1, \ldots, v_p) \in \mathbb{N}^p$ 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 $v_i \in \mathbb{N}$ 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 ($+, -, \times$) that are safe in the sense that they are defined on all possible inputs in $\mathbb{N}^2$.
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\textsuperscript{\textregistered} 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\textsuperscript{\textregistered} 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:

\begin{verbatim}
Variables : x_1, ..., x_p
Return Var : x_r
Constants : n_1, ..., n_q

ProgName(n_1, ..., n_q)
{
   [CODE]
}
\end{verbatim}

where $x_1, ..., x_p$ are the renamed local variables of the program. $n_1, ..., n_q$ 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:

\begin{verbatim}
int Fac(int n)
{
   int res = 1;
   int ctr = 0;
\end{verbatim}
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.

91
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 \texttt{func} that was presented in chapter 1. we used Nivea to create the Model. The program \texttt{func} was defined as follows:

\begin{verbatim}
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;
    }
}
\end{verbatim}

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

\begin{verbatim}
Vars :x_1 x_2 x_3
Return Var :
Constants : n_1, n_2
\end{verbatim}

\texttt{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;
\end{verbatim}

92
\[2\] x_2 = n_1 \ast x_2;  
\[1\] x_3 = x_3 \ast 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

\[
\begin{align*}
\text{Constants} : & n_1, n_2 \\
(Mf) \forall (x_1, x_2, x_3) : & \text{func}(x_1, x_2, x_3, n_1, n_2) = Wfunc1(Afunc4(Afunc5(Afunc6(x_1, x_2, x_3, n_1, n_2))), w) \\
\Gamma_M : \\
\text{Defining} : & Afunc1 \text{ from assignment statement} : [1] x_3 = x_2 \ast x_1; \\
[Mf1] \forall (x_1, x_2, x_3) : & (Afunc1(x_1, x_2, x_3, n_1, n_2)) = (x_1, x_2, (x_3 \ast x_1), n_1, n_2) \\
[Mf2] \forall (x_1, x_2, x_3) : & (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3) = Afunc1(x_1, x_2, x_3, n_1, n_2))) \\
\text{End Definition} (Afunc1) \\
\text{Defining} : & Afunc2 \text{ from assignment statement} : [2] x_2 = n_1 \ast x_2; \\
[Mf3] \forall (x_1, x_2, x_3) : & (Afunc2(x_1, x_2, x_3, n_1, n_2)) = (x_1, (n_1 \ast x_2), x_3, n_1, n_2) \\
[Mf4] \forall (x_1, x_2, x_3) : & (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3) = Afunc2(x_1, x_2, x_3, n_1, n_2))) \\
\text{End Definition} (Afunc2) \\
\text{Defining} : & Afunc3 \text{ from assignment statement} : [3] x_1 = x_1 + 1; \\
[Mf5] \forall (x_1, x_2, x_3) : & (Afunc3(x_1, x_2, x_3, n_1, n_2)) = ((x_1 + 1), x_2, x_3, n_1, n_2) \\
[Mf6] \forall (x_1, x_2, x_3) : & (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3) = Afunc3(x_1, x_2, x_3, n_1, n_2))) \\
\text{End Definition} (Afunc3)
\end{align*}
\]

93
Defining: $W_{\text{func1}}$ from while statement ::-while($x_1 < n_2$){...}::-

$[Mf7] \forall (x_1, x_2, x_3, w) : ((0 = w) \rightarrow (W_{\text{func1}}(x_1, x_2, x_3, n_1, n_2, w) = (x_1, x_2, x_3, n_1, n_2)))$

$[Mf8] \forall (x_1, x_2, x_3, w) :$

$((0 < w) \rightarrow (W_{\text{func1}}(x_1, x_2, x_3, n_1, n_2, w) = A_{\text{func1}}(A_{\text{func2}}(A_{\text{func3}}(W_{\text{func1}}(x_1, x_2, x_3, n_1, n_2, (w - 1)))))))$

$[Mf9] \forall (x_1, x_2, x_3, w) : (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3, n_1, n_2) = W_{\text{func1}}(x_1, x_2, x_3, n_1, n_2, w)))$

End Definition ($W_{\text{func1}}$)

Defining: $A_{\text{func4}}$ from assignment statement ::-[4]$x_3 = 1;$::-

$[Mf10] \forall (x_1, x_2, x_3) : (A_{\text{func4}}(x_1, x_2, x_3, n_1, n_2) = (x_1, x_2, (1), n_1, n_2))$

$[Mf11] \forall (x_1, x_2, x_3) : (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3) = A_{\text{func4}}(x_1, x_2, x_3, n_1, n_2)))$

End Definition ($A_{\text{func4}}$)

Defining: $A_{\text{func5}}$ from assignment statement ::-[5]$x_2 = 1;$::-

$[Mf12] \forall (x_1, x_2, x_3) : (A_{\text{func5}}(x_1, x_2, x_3, n_1, n_2) = (x_1, (1), x_3, n_1, n_2))$

$[Mf13] \forall (x_1, x_2, x_3) : (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3) = A_{\text{func5}}(x_1, x_2, x_3, n_1, n_2)))$

End Definition ($A_{\text{func5}}$)

Defining: $A_{\text{func6}}$ from assignment statement ::-[6]$x_1 = 0;$::-

$[Mf14] \forall (x_1, x_2, x_3) : (A_{\text{func6}}(x_1, x_2, x_3, n_1, n_2) = ((0), x_2, x_3, n_1, n_2))$

$[Mf15] \forall (x_1, x_2, x_3) : (\exists (y_1, y_2, y_3) : ((y_1, y_2, y_3) = A_{\text{func6}}(x_1, x_2, x_3, n_1, n_2)))$

End Definition ($A_{\text{func6}}$)

Remark 6.3.1 In the definition of $\text{func}$, we added $w$ as an unquantified variable. It is just a place holder that is used to keep the arity of $W_{\text{func1}}$ 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:

$\forall (x_1, x_2, x_3) : \exists (w) :$

$\text{func}(x_1, x_2, x_3, n_1, n_2) = W_{\text{func1}}(A_{\text{func4}}(A_{\text{func5}}(A_{\text{func6}}(x_1, x_2, x_3, n_1, n_2))), w)$

94
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 \textsc{func}.

\textbf{Proof 1 :}

We then set about to prove some properties of the \textsc{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 \([D1]:\)

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

\[
(<(y_1, y_2, y_3, n_1, n_2) = \text{Afunc1}(\text{Afunc2}(\text{Afunc3}(x_1, x_2, x_3, n_1, n_2)))) > \\
\land(< x_3 = \text{fac}(x_1) > \land < x_2 = \text{exp}(n_1, x_1) >) \\
\rightarrow < y_3 = \text{fac}(y_1) > \land < y_2 = \text{exp}(n_1, y_1) >
\]

\textbf{Proof of \([D1]:\)}

\begin{tabular}{l}
1. \textbf{[Variables]}(y_1, y_2, y_3, x_1, x_2, x_3) \\
2. \textbf{[Assumption]}(<(y_1, y_2, y_3, n_1, n_2) = \text{Afunc1}(\text{Afunc2}(\text{Afunc3}(x_1, x_2, x_3, n_1, n_2)))) > \\
\land(< x_3 = \text{fac}(x_1) > \land < x_2 = \text{exp}(n_1, x_1) >) \\
3. \textbf{[\land Elim 2]}<(y_1, y_2, y_3, n_1, n_2) = \text{Afunc1}(\text{Afunc2}(\text{Afunc3}(x_1, x_2, x_3, n_1, n_2))) > \\
4. \textbf{[\land Elim 2]}(< x_3 = \text{fac}(x_1) > \land < x_2 = \text{exp}(n_1, x_1) >) \\
5. \textbf{[Mf3]} \\
6. \textbf{[\forall Elim 5]}< \text{Afunc3}(x_1, x_2, x_3, n_1, n_2) = ((x_1 + 1).x_2.x_3.n_1.n_2) > \\
7. \textbf{[= Sub 6. 3]}<(y_1, y_2, y_3, n_1, n_2) = \text{Afunc1}(\text{Afunc2}(x_1 + 1).x_2.x_3.n_1.n_2) > \\
8. \textbf{[Mf3]} \\
9. \textbf{[\forall Elim 8]}< \text{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. \textbf{[= Sub 9. 7]}<(y_1, y_2, y_3, n_1, n_2) = \text{Afunc1}(x_1 + 1).n_1.x_2.x_3.n_1.n_2 > \\
11. \textbf{[Mf1]} \\
12. \textbf{[\forall Elim 11]}< \text{Afunc1}(x_1 + 1).n_1.x_2.x_3.n_1.n_2 > \\
\textbf{=} ((x_1 + 1).n_1.x_2.x_3.n_1.n_2 > \\
13. \textbf{[= Sub 12. 10]}<(y_1, y_2, y_3, n_1, n_2) = ((x_1 + 1).n_1.x_2.x_3.n_1.n_2)>
\end{tabular}
14. \[=\text{Elim 13}\] \(< y_1 = x_1 + 1 >\)
15. \[=\text{Elim 13}\] \(< y_2 = n_1 \cdot z_2 >\)
16. \[=\text{Elim 13}\] \(< y_3 = (x_3 \cdot (x_1 + 1)) >\)
17. \[\text{Number Theory}\] \(<\forall(z_1, z_2) : z_1 = \text{fac}(z_2) >\)
   \[\rightarrow z_1 \cdot (z_2 + 1) = \text{fac}(z_2 + 1) >\)
18. \[\forall\text{ Elim 17}\] \(< x_3 = \text{fac}(x_1) >\)
   \[\rightarrow x_3 \cdot (x_1 + 1) = \text{fac}(x_1 + 1) >\)
19. \[=\text{Sub 16, 18}\] \(< x_3 = \text{fac}(x_1) >\)
   \[\rightarrow < y_3 = \text{fac}(x_1 + 1) >\)
20. \[=\text{Sub 14, 19}\] \(< x_3 = \text{fac}(x_1) >\)
   \[\rightarrow < y_3 = \text{fac}(y_1) >\)
21. \[\land\text{ Elim 4}\] \(< x_3 = \text{fac}(x_1) >\)
22. \[=\text{Elim 20, 21}\] \(< y_3 = \text{fac}(y_1) >\)
23. \[\text{Number Theory}\] \(<\forall(z_1, z_2) : z_1 = \text{exp}(n_1, z_2) >\)
   \[\rightarrow < n_1 \cdot z_1 = \text{exp}(n_1, z_2 + 1) >\)
24. \[\forall\text{ Elim 23}\] \(< x_2 = \text{exp}(n_1, x_1) >\)
   \[\rightarrow < n_1 \cdot x_2 = \text{exp}(n_1, x_1 + 1) >\)
25. \[=\text{Sub 14, 24}\] \(< x_2 = \text{exp}(n_1, x_1) >\)
   \[\rightarrow < n_1 \cdot x_2 = \text{exp}(n_1, y_1) >\)
26. \[=\text{Sub 15, 25}\] \(< x_2 = \text{exp}(n_1, x_1) >\)
   \[\rightarrow < y_2 = \text{exp}(n_1, y_1) >\)
27. \[\land\text{ Elim 4}\] \(< x_2 = \text{exp}(n_1, x_1) >\)
28. \[=\text{Elim 27, 26}\] \(< y_2 = \text{exp}(n_1, y_1) >\)
29. \[\land\text{ Intro 22, 28}\] \(< y_3 = \text{fac}(y_1) >\)
   \[\land < y_2 = \text{exp}(n_1, y_1) >\)
30. \[=\text{Intro 2, 3-29}\] \(< (y_1, y_2, y_3, n_1, n_2) = Afunc1(Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2)) >\)
   \[\land(x_3 = \text{fac}(x_1) >\)
   \[\land < x_2 = \text{exp}(n_1, x_1) >\)
   \[\rightarrow < y_3 = \text{fac}(y_1) >\)
   \[\land < y_2 = \text{exp}(n_1, y_1) >\)
27. \[\forall\text{ Intro 1, 2-30}\]
   \(<\forall(x_1, x_2, x_3) : Wfunc1(x_1, x_2, x_3, n_1, n_2, 0) = (x_1, x_2, x_3, n_1, n_2) >\)

Proof 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. \[ \forall \text{ Elim 1} \forall(x_1, x_2, x_3) : (<0 = 0>) \]
   \[ \quad \rightarrow (<\text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) = (x_1, x_2, x_3, n_1, n_2)>) \]
3. [Vars](z_1, z_2, z_3)
4. \[ \forall \text{ Elim 2} < (0 = 0) > \]
   \[ \rightarrow <\text{Wfunc1}(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) > \]
5. [Number Theory] < 0 = 0 >
6. \[ \forall \text{ Elim 4,5} <\text{Wfunc1}(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) > \]
7. \[ \forall \text{ Intro 3, 4,6} \forall(z_1, z_2, z_3) : \]
   \[ <\text{Wfunc1}(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2) > \]

Proof 3:

We now want to prove a property that comes from the while loop part of the code. Since the \text{Wfunc1} function that was generated by the \text{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]:

\[ \forall(y_1, y_2, y_3, x_1, x_2, x_3) : \]
\[ (<(y_1, y_2, y_3, n_1, n_2) = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) > \]
\[ \land (<x_3 = \text{fac}(x_1) > \land <x_2 = \text{exp}(n, x_1)>) \]
\[ \rightarrow (<y_3 = \text{fac}(y_1) > \land <y_2 = \text{exp}(n, y_1)>) \]

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) = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) >
   \[ \land (<x_3 = \text{fac}(x_1) > \land <x_2 = \text{exp}(n, x_1)>) \]
3. [D2 RENAME](z_1, z_2, z_3) : <\text{Wfunc1}(z_1, z_2, z_3, n_1, n_2, 0) = (z_1, z_2, z_3, n_1, n_2)>
4. [\forall \text{ Elim 3}] <\text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) = (x_1, x_2, x_3, n_1, n_2) >
5. [\land \text{ Elim 2}] <\text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) >
6. [\text{=Sub 5,4}] <(y_1, y_2, y_3, n_1, n_2) = (x_1, x_2, x_3, n_1, n_2) >
7. [\text{=Elim 6}] < y_1 = x_1 >
8. [\text{=Elim 6}] < y_2 = x_2 >
9. \[=\text{Elim 6}] (y_3 = x_3 >
10. \[=\text{Elim 2}] (x_3 = \text{fac}(x_1) > \land x_2 = \text{exp}(n, x_1) >)
11. \[=\text{Sub 10.7}] (x_3 = \text{fac}(y_1) > \land x_2 = \text{exp}(n, y_1) >)
12. \[=\text{Sub 11.8}] (x_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n, y_1) >)
13. \[=\text{Sub 12.9}] (y_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n, y_1) >)
14. \[\text{Intro 2, 3-13}] (x_3 = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) >
\land (x_3 = \text{fac}(x_1) > \land x_2 = \text{exp}(n, x_1) >))
\[\text{Intro 1, 2-14}] (x_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n, y_1) >)
15. \[=\text{Intro 1, 2-14}] (x_3 = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, 0) >
\land (x_3 = \text{fac}(x_1) > \land y_2 = \text{exp}(n, x_1) >))
\[\text{Intro 1, 2-14}] (x_3 = \text{fac}(y_1) > \land y_2 = \text{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):
(\langle(y_1, y_2, y_3, n_1, n_2) = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, \text{ind} - 1) > \land (x_3 = \text{fac}(x_1) > \land x_2 = \text{exp}(n_1, x_1) >))
\[\text{Intro 1, 2-14}] (x_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n_1, y_1) >)))
\[\text{Intro 1, 2-14}] (x_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n_1, y_1) >)))
\]

Proof of [D4]:

1. [Assumption] \[\forall(y_1, y_2, y_3, x_1, x_2, x_3):
(\langle(y_1, y_2, y_3, n_1, n_2) = \text{Wfunc1}(x_1, x_2, x_3, n_1, n_2, \text{ind} - 1)
\land (x_3 = \text{fac}(x_1) > \land x_2 = \text{exp}(n_1, x_1) >))
\[\text{Intro 1, 2-14}] (x_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n_1, y_1) >)
\]
2. [Variables] \[\forall(z_1, z_2, z_3, w_1, w_2, w_3):
(\langle(z_1, z_2, z_3, n_1, n_2) = \text{Wfunc1}(w_1, w_2, w_3, n_1, n_2, \text{ind})
\land (w_3 = \text{fac}(w_1) > \land w_2 = \text{exp}(n_1, w_1) >))
\]
3. [Number Theory] \[\forall(x) : (\langle 0 < x > \[\text{Intro 1, 2-14}] (\langle 0 = x - 1 > \lor 0 < x - 1 >))
\]

98
5. $\forall$ Elim 4($< 0 < \text{ind} > \rightarrow (< 0 = \text{ind} - 1 > \lor < 0 < \text{ind} - 1 >)$)
6. [Induction 1] $< 0 < \text{ind} >$
7. $\forall$ Elim 6, 5 $< 0 = \text{ind} - 1 > \lor < 0 < \text{ind} - 1 >$
8. [And Elim 3] $<(z_1, z_2, z_3, n_1, n_2) = \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind}) >$
9. [Mf8]
10. $\forall$ Elim 9($< 0 < \text{ind} >$) $\rightarrow$
$< \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind}) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind} - 1)))) >$
11. $\forall$ Elim 6, 10
$< \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind}) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind} - 1)))) >$
12. [Assumption] $< 0 = \text{ind} - 1 >$
13. $[\forall$ Elim 11, 12]
$< \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind}) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2, 0)))) >$
14. $[D2]\forall(y_1, y_2, y_3) < \text{Wfunc}(y_1, y_2, y_3, n_1, n_2, 0) = (y_1, y_2, y_3, n_1, n_2) >$
15. $[\forall$ Elim 14] $< \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, 0) = (w_1, w_2, w_3, n_1, n_2) >$
16. $[\forall$ Elim 15, 13]
$< \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind}) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2)) >$
17. $[\forall$ Elim 3] $< (z_1, z_2, z_3, n_1, n_2) = \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, \text{ind}) >$
18. $[\forall$ Elim 17, 16] $< (z_1, z_2, z_3, n_1, n_2) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2)) >$
19. $[\forall$ Elim 3] $< w_3 = \text{fac}(w_1) > \land < w_2 = \exp(n_1, w_1) >$
20. $[\forall$ Intro 18, 19]
$<(z_1, z_2, z_3, n_1, n_2) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2)) >$
$\land < w_3 = \text{fac}(w_1) > \land < w_2 = \exp(n_1, w_1) >)$
21. $[D1]\forall(y_1, y_2, y_3, x_1, x_2, x_3)$:
$<(x_1 = \text{fac}(x_1) > \land < x_2 = \exp(n_1, x_1) >)$
$<(y_1 = \text{fac}(y_1) > \land < y_2 = \exp(n_1, y_1) >)$
22. $[\forall$ Elim 21] $<(z_1, z_2, z_3, n_1, n_2) = \text{Afunc}(\text{Afunc}(\text{Wfunc}(w_1, w_2, w_3, n_1, n_2)) >$
$\land < w_3 = \text{fac}(w_1) > \land < w_2 = \exp(n_1, w_1) >)$
$<(z_3 = \text{fac}(z_1) > \land < z_2 = \exp(n_1, z_1) >)$
500. $\forall$ Elim 20, 22
$<(z_3 = \text{fac}(z_1) > \land < z_2 = \exp(n_1, z_1) >)$
501. [Assumption] $< 0 < \text{ind} - 1 >$
502. [Mf9]
503. [Mf9] $\exists(v_1, v_2, v_3 : (v_1, v_2, v_3, n_1, n_2) = \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, (\text{ind} - 1))$
504. [Variables] $(v_0^0, v_0^2, v_0^3), (v_0^0, v_0^2, v_0^3, n_1, n_2) = \text{Wfunc}(w_1, w_2, w_3, n_1, n_2, (\text{ind} - 1))$
505. \[ \forall \text{ Elim 1}\{ (v^0_1, v^0_2, v^0_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, (ind - 1)) > \]
\[\land (w_3 = \text{fac}(w_1) > \land w_2 = \text{exp}(n_1, w_1) >) \]
\[\rightarrow (v^0_2 = \text{fac}(v^0_1) > \land v^0_2 = \text{exp}(n_1, v^0_1) >)\]

506. \[\text{And Elim 3}\{(w_3 = \text{fac}(w_1) > \land w_2 = \text{exp}(n_1, w_1) >)\]

507. \[\text{And Intro 504, 506}\{(v^0_1, v^0_2, v^0_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, (ind - 1)) > \]
\[\land (w_3 = \text{fac}(w_1) > \land w_2 = \text{exp}(n_1, w_1) >) \]
\[\rightarrow \text{ Elim 507, 505}\{(v^0_1 = \text{fac}(v^0_1) > \land v^0_2 = \text{exp}(n_1, v^0_1) >)\]

509. \[\text{Sub 504, 11}\]
\[Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) = Afunc2(Afunc3(v^0_1, v^0_2, v^0_3, n_1, n_2) > \]
\[\text{Sub 8, 509}\{(z_1, z_2, z_3, n_1, n_2) = Afunc2(Afunc3(v^0_1, v^0_2, v^0_3, n_1, n_2) > \]

511. \[\text{D1}\{y_1, y_2, y_3, x_1, x_2, x_3\} : \]
\[\{(y_1, y_2, y_3, n_1, n_2) = Afunc2(Afunc3(x_1, x_2, x_3, n_1, n_2) > \]
\[\land (x_3 = \text{fac}(x_1) > \land x_2 = \text{exp}(n_1, x_1) >) \]
\[\rightarrow (y_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n_1, y_1) >)\]

512. \[\forall \text{ Elim 511}\{(z_1, z_2, z_3, n_1, n_2) = Afunc2(Afunc3(v^0_1, v^0_2, v^0_3, n_1, n_2) > \]
\[\land (v^0_1 = \text{fac}(v^0_1) > \land v^0_2 = \text{exp}(n_1, v^0_1) >) \]
\[\rightarrow (z_3 = \text{fac}(z_1) > \land z_2 = \text{exp}(n_1, z_1) >)\]

513. \[\land \text{ Intro 510, 508}\{(z_1, z_2, z_3, n_1, n_2) = Afunc2(Afunc3(v^0_1, v^0_2, v^0_3, n_1, n_2) > \]
\[\land (v^0_1 = \text{fac}(v^0_1) > \land v^0_2 = \text{exp}(n_1, v^0_1) >) \]

994. \[\rightarrow \text{ Elim 512, 513}\{(z_3 = \text{fac}(z_1) > \land z_2 = \text{exp}(n_1, z_1) >)\]

995. \[\exists \text{ Elim 503, 504-994}\{(z_3 = \text{fac}(z_1) > \land z_2 = \text{exp}(n_1, z_1) >)\]

996. \[\forall \text{ Elim 7, 12-500, 501-995}\{(z_3 = \text{fac}(z_1) > \land z_2 = \text{exp}(n_1, z_1) >)\]

997. \[\rightarrow \text{ Intro 3, 4-996}\{(z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) > \]
\[\land (w_3 = \text{fac}(w_1) > \land w_2 = \text{exp}(n_1, w_1) >) \]
\[\rightarrow (z_3 = \text{fac}(z_1) > \land z_2 = \text{exp}(n_1, z_1) >)\]

998. \[\forall \text{ Intro 2-997}\{z_1, z_2, z_3, w_1, w_2, w_3\} : \]
\[\{(z_1, z_2, z_3, n_1, n_2) = Wfunc1(w_1, w_2, w_3, n_1, n_2, ind) > \]
\[\land (w_3 = \text{fac}(w_1) > \land w_2 = \text{exp}(n_1, w_1) >) \]
\[\rightarrow (z_3 = \text{fac}(z_1) > \land z_2 = \text{exp}(n_1, z_1) >)\]

999. \[\text{Rename}\{y_1, y_2, y_3, x_1, x_2, x_3\} : \]
\[\{(y_1, y_2, y_3, n_1, n_2) = Wfunc1(x_1, x_2, x_3, n_1, n_2, ind) > \]
\[\land (x_3 = \text{fac}(x_1) > \land x_2 = \text{exp}(n_1, x_1) >) \]
\[\rightarrow (y_3 = \text{fac}(y_1) > \land y_2 = \text{exp}(n_1, y_1) >)\]
\[ \forall(y_1, y_2, y_3, x_1, x_2, x_3) : \\
\langle (y_1, y_2, y_3, n_1, n_2) = W\text{func1}(x_1, x_2, x_3, n_1, n_2, ind - 1) > \\
\land \langle x_3 = \text{fac}(x_1) > \land < x_2 = \text{exp}(n_1, x_1) > \rangle \\
\rightarrow (\langle y_3 = \text{fac}(y_1) > \land < y_2 = \text{exp}(n_1, y_1) > \rangle) \\
\] 

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]:

\[ \forall(w) : \forall(y_1, y_2, y_3, x_1, x_2, x_3) : \\
\langle (y_1, y_2, y_3, n_1, n_2) = W\text{func1}(x_1, x_2, x_3, n_1, n_2, w) > \\
\land \langle x_3 = \text{fac}(x_1) > \land < x_2 = \text{exp}(n, x_1) > \rangle \\
\rightarrow (\langle y_3 = \text{fac}(y_1) > \land < y_2 = \text{exp}(n, y_1) > \rangle) \\
\]
Proof of [D5]:

1. [D3] ∀(y₁, y₂, y₃, x₁, x₂, x₃) :
   (<(y₁, y₂, y₃, n₁, n₂) = Wfuncl(x₁, x₂, x₃, n₁, n₂, 0) >
   ∧(<x₃ = fac(x₁) > ∧ <x₂ = exp(n, x₁) >))
   → (<y₂ = fac(y₁) > ∧ <y₂ = exp(n, y₁) >)

2. [D4] ∀(y₁, y₂, y₃, x₁, x₂, x₃) :
   (<>(y₁, y₂, y₃, n₁, n₂) = Wfuncl(x₁, x₂, x₃, n₁, n₂, ind - 1) >
   ∧(<x₃ = fac(x₁) > ∧ <x₂ = exp(n, x₁) >))
   → (<y₂ = fac(y₁) > ∧ <y₂ = exp(n, y₁) >))
   →
   (∀(y₁, y₂, y₃, x₁, x₂, x₃) :
   (<>(y₁, y₂, y₃, n₁, n₂) = Wfuncl(x₁, x₂, x₃, n₁, n₂, ind) >
   ∧(<x₃ = fac(x₁) > ∧ <x₂ = exp(n, x₁) >))
   → (<y₂ = fac(y₁) > ∧ <y₂ = exp(n, y₁) >))

3. [Induction 1. 2] ∀(w) : ∀(y₁, y₂, y₃, x₁, x₂, x₃) :
   (<>(y₁, y₂, y₃, n₁, n₂) = Wfuncl(x₁, x₂, x₃, n₁, n₂, w) >
   ∧(<x₃ = fac(x₁) > ∧ <x₂ = exp(n, x₁) >))
   → (<y₂ = fac(y₁) > ∧ <y₂ = exp(n, y₁) >)

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:

∀(y₁, y₂, y₃, x₁, x₂, x₃) :
<>(y₁, y₂, y₃, n₁, n₂) = func(x₁, x₂, x₃, n₁, n₂) >
→ (<y₂ = fac(y₁) > ∧ <y₂ = exp(n, y₁) >)

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) = \text{func}(x_1, x_2, x_3, n_1, n_2)\>
3. [Mf]  
4. \([\forall \text{Elim} 3]\) \(\text{func}(x_1, x_2, x_3, n_1, n_2) \implies W\text{func1}(\text{Afunc4}(\text{Afunc5}(\text{Afunc6}(x_1, x_2, x_3, n_1, n_2)), w))\)
5. \([= \text{Sub} 2, 4]\) \(<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}(\text{Afunc4}(\text{Afunc5}(\text{Afunc6}(x_1, x_2, x_3, n_1, n_2)), w))\>
6. [Mf14]  
7. \([\forall \text{Elim}, 6]\) \(<\text{Afunc6}(x_1, x_2, x_3, n_1, n_2) = ((0). x_2, x_3, n_1, n_2)\>
8. \([= \text{Sub} 5, 7]\) \(<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}(\text{Afunc4}(\text{Afunc5}((0). v_2, v_3, n_1, n_2), w))\>
9. [Mf12]  
10. \([\forall \text{Elim}, 9]\) \(<\text{Afunc5}((0). x_2, x_3, n_1, n_2) = ((0). (1). x_3, n_1, n_2)\>
11. \([= \text{Sub} 8, 10]\) \(<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}(\text{Afunc4}((0). (1). x_3, n_1, n_2), w)\>
12. [Mf10]  
13. \([\forall \text{Elim}, 12]\) \(<\text{Afunc4}((0). (1). x_3, n_1, n_2) = ((0). (1). (1). n_1, n_2)\>
14. \([= \text{Sub} 11, 13]\) \(<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}((0, 1, 1, n_1, n_2). w)\>
15. \([\text{collapse} 14]\) \(<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}(0, 1, 1, n_1, n_2, w)\>
16. [Number Theory] \(<1 = \text{fac}(0)\>
17. [Number Theory] \(<1 = \text{exp}(n_1, 0)\>
18. \([\land \text{Intro}, 16, 17]\) \(<1 = \text{fac}(0)\land (1 = \text{exp}(n_1, 0))\>
19. \([\land \text{Intro}, 15, 18]\) \(<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}(0, 1, 1, n_1, n_2, w)\land (1 = \text{fac}(0)\land (1 = \text{exp}(n_1, 0))\>
20. [D5 Rename] \:<(w_1, v_1, v_2, v_3, w_1, w_2, w_3):\>
\(<(w_1, w_2, w_3, n_1, n_2) = W\text{func1}(w_1, w_2, w_3, n_1, n_2, w)\land (w_3 = \text{fac}(w_1)\land (w_2 = \text{exp}(n, w_1))\>
\(<v_3 = \text{fac}(v_1)\land (v_2 = \text{exp}(n, v_1))\>
\(<(y_3 = \text{fac}(y_1))\land (y_2 = \text{exp}(n, y_1))\>
21. \([\forall \text{Elim} 20]\) \:<(y_1, y_2, y_3, n_1, n_2) = W\text{func1}(0, 1, 1, n_1, n_2, w)\land (1 = \text{fac}(0)\land (1 = \text{exp}(n, 0))\>
\(<(y_3 = \text{fac}(y_1)\land (y_2 = \text{exp}(n, y_1))\>
22. \([\rightarrow \text{Intro}, 19, 21]\) \(<y_3 = \text{fac}(y_1)\land (y_2 = \text{exp}(n, y_1))\>
23. \([\rightarrow \text{Intro}, 2, 3-22]\) \:<(y_1, y_2, y_3, n_1, n_2) = \text{func}(x_1, x_2, x_3, n_1, n_2)\>
\(<(y_3 = \text{fac}(y_1)\land (y_2 = \text{exp}(n, y_1))\>
24. [\forall \text{Intro 1, 2-23}] \:<(y_1, y_2, y_3, x_1, x_2, x_3):\>
\(<(y_1, y_2, y_3, n_1, n_2) = \text{func}(x_1, x_2, x_3, n_1, n_2)\>
\(<(y_3 = \text{fac}(y_1)\land (y_2 = \text{exp}(n, y_1))\>)\>

103
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:

```lisp
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 ‘deftm’ 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 !> (deftm 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

\[(\text{AND} (\text{IMPLIES} (\text{NOT} (\text{CONSP} X)) (:P X Y))\]
\[ (\text{IMPLIES} (\text{AND} (\text{CONSP} X) (:P (\text{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
\[(\text{IMPLIES} (\text{NOT} (\text{CONSP} X))\]
\[ (\text{EQUAL} (\text{LEN} (\text{MY-APP} X Y))
\[ (+ (\text{LEN} X) (\text{LEN} Y))).\]

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
\[(\text{IMPLIES} (\text{AND} (\text{CONSP} X))\]
\[ (\text{EQUAL} (\text{LEN} (\text{MY-APP} (\text{CDR} X) Y))
\[ (+ (\text{LEN} (\text{CDR} X)) (\text{LEN} Y))))
\[ (\text{EQUAL} (\text{LEN} (\text{MY-APP} X Y))
\[ (+ (\text{LEN} X) (\text{LEN} Y))).\]

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  

108
;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 existence 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>  ==>  <AFaci(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))))

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

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

;=================================================================================
;=================================================================================

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))
     )
)

112
(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, \textit{GFac1} is defined as:

\begin{verbatim}
(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)))
  )
\end{verbatim}

To which ACL2 replies:

Since \textit{GFAC1} is non-recursive, its admission is trivial. We observe that the type of \textit{GFAC1} is described by the theorem (AND (CONSP (GFAC1 ARGUMENTS)) (TRUE-LISTP (GFAC1 ARGUMENTS))). We used the :type-prescription rule \textit{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 \textit{GFac2}:

\begin{verbatim}
(defun GFac2 (arguments)
  (let
  \end{verbatim}
(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 WFacia (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(WFacia (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 EO-ORD-< (which is known to be well-founded on the domain recognized by EO-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-PRESRIPTION 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 EO-ORD-< (which is known to be well-founded on the domain recognized by EO-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 ...)

116
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)

We start by proving the equivalent of the $S_L$ expression:

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

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:  ( DEFTM 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, fibonacci,...). 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 fibonacci program and of a fibonacci 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:

\[ \varphi(A) = x_1 \geq 0 \land x_2 > 0 \]
\[ \varphi(B) = x_1 \geq 0 \land x_2 > 0 \land y_1 \equiv 0 \]
\[ \varphi(C) = (x_1 \equiv y_1 \cdot x_2 + y_2) \land y_2 \geq 0 \]
\[ \varphi(D) = (x_1 \equiv y_1 \cdot x_2 + y_2) \land y_2 \geq x_2 \]
\[ \varphi(E) = (x_1 \equiv y_1 \cdot x_2 + y_2 - x_2) \land y_2 - x_2 \geq 0 \]
\[ \varphi(F) = (x_1 \equiv y_1 \cdot x_2 + y_2 - x_2) \land y_2 \geq 0 \land y_2 < x_2 \]

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. \( \text{pre}(c) \) is the predicate that holds before \( c \) and \( \text{post}(c) \) is the predicate that holds after \( c \)):

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

  \[ (\text{pre}(c) \land p) \rightarrow \text{post}(c) \]

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

- If \( c \) is a negative decision,

  \[ (\text{pre}(c) \land \neg p) \rightarrow \text{post}(c) \]

  needs to be proven (F arrow).

- If \( c \) is an assignment,

  \[ \text{pre}(c) \rightarrow \text{post}(c)[e/v] \]

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

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

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

124
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:

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

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*. 

126
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. [∀ 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. [∀ 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. [∀ 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. [→ Intro 2.3-11] \(< (y_1, y_2, y_3, n_1, n_2) = Afunc4(Afunc5(Afunc6(v_1, v_2, v_3, n_1, n_2))) >\)

\[< (y_1, y_2, y_3, n_1, n_2) = (0, 1, 1, n_1, n_2) >\]

13. [∀ Intro 1.2-12] \[∀ (y_1, y_2, y_3, v_1, v_2, v_3) : \]

\[< (y_1, y_2, y_3, n_1, n_2) = Afunc4(Afunc5(Afunc6(v_1, v_2, v_3, n_1, n_2))) >\]

\[< (y_1, y_2, y_3, n) = (0, 1, 1, n_1, n_2) >\]

• This also means that if the program is changed, it is easier to keep and 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:

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

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

This would formally translate in $S_L$ to

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

Where $F_{Cond} : \mathbb{N} \rightarrow \mathbb{B}$ 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


Appendix A

ACL2 output for the main proof of Chapter 6

ACL2 Warning [Non-rec] in ( DEPTHM 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)))

131
(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-preservation 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 EO-ORD-< (which is known to be well-founded on the domain recognized by EO-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)))

134
(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'
(INPLIES (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)))

136
(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)

137
(* 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)))

138
(EQUAL (FAC (+ 1 TUPLE3))
  (+ I (* TUPLE3 I))).

This simplifies, using the :rewrite rule COMUTATIVITY-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.

140
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))
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 EO-ORD<- (which is known to be well-founded on the domain recognized by EO-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-WTH 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 BINAR Y-, INTEGER, 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)
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

148
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 implementation 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)
```

151
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($Statement =~ m/^\s*([\d\s]+)\s*$/) {
        $Statement = $1;
        $Return = ProcessAssignment($Statement);
    }

    #Is It While
    elsif($Statement =~ m/^\s*(while.*)\s*$/) {
        $Statement =~ m/^\s*$/;
        $Return = ProcessWhile($Statement);
    }

    #Is It If
    elsif($Statement =~ m/^\s*(if.*)\s*$/) {
        $Statement =~ m/^\s*$/;
        $Return = ProcessIf($Statement);
    }

    else {
        $Return = $Statement
    }

    153
B.3 Code for the Process Assignment Implementation

my @AssignmentFunctionVariables;
my @AssignmentFunctions;

# Processes Assignment
sub ProcessAssignment
{
    my $AssignmentStatement = $[_0];
    my $Return;

    if($AssignmentStatement =~ m/^\s*(\[\d+\].*)\s*$/) {
        $AssignmentStatement = $1;
        $Return = MakeAssignment($AssignmentStatement);
    } else {
        $Return = $AssignmentStatement
            ."\$AssignmentStatement:";
    }
    $Return;
}
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\n";
$AssignmentStatement = " m/"\[(\d+)](\w_\d)(\s*)=(\s*)(.+);(\s*)$/; $FuncBody = &replace($2, "("", "$.")", $FuncBody);

#Creating Function definitions
$Funcdef = $Funcdef."\\forall (" .join("", @ProgVariables)."):";
$Funcdef = $Funcdef."$.Return
 ."(".join("", @ProgVariables)
 .join("", @ProgConstants).") = (";
$Funcdef = $Funcdef.join("", @FuncBody)
 .join("", @ProgConstants)."))\\\\\n";

$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)";

156
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 $WhileLoopConst = "1"."_".$WhileLoopNum;
    $WhileLoopConsts = $WhileLoopConsts." ".$WhileLoopConst;

    "--------------------------------------------------------------------------------
    #While A Function (General function in partial completeness)
    "--------------------------------------------------------------------------------

    160
my $WFuncAName = $Return;

my $WFuncDef = "$\n\n\nDefining : $\n\n$ from while statement : -\\\emph\{while\}$
.$CondStatement."\\\{dots\}\{\}$-\\\n";

$WFuncDef = $WFuncDef."\\\forall\{.join"," , @ProgVariables) . \},w):";
$WFuncDef = $WFuncDef."(0 = w)\\implies(".$WFuncAName
."(".$StdArgs.\",w) = (".$StdArgs.\")))\\n"
$WFuncDef = $WFuncDef."\\\forall\{.join"," , @ProgVariables) . \},w):";
$WFuncDef = $WFuncDef."(0 < w)\\implies(".$WFuncAName
."(".$StdArgs.\",w) = ";

#Quick Hack
my $TempArgs = $WFuncAName."(".$StdArgs.\",(w - 1))";
$BodyCodeNameTemp = $BodyCodeName;
$BodyCodeNameTemp = s/(/((\w+,?)\+?)/(\$TempArgs)/;
$WFuncDef = $WFuncDef.$BodyCodeNameTemp."))\\n";
$Model = $Model.$WFuncDef;

my $WFuncAInfo = "$\\\forall\{.join"," , @ProgVariables) . \},w):
(\exists\{.join(".,@ProgVariablesSub).\}):";
$WFuncAInfo = $WFuncAInfo."(".$join"," ,@ProgVariablesSub)."=".$WFuncAName."(".$StdArgs.\",w))";

$Model = $Model.$WFuncAInfo;
$Model = $Model."End Definition $"
#While B Function (Completeness 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;

}