Towards software certification. Encoding of an imperative language in the meta-logical framework LF.

Jesús Domínguez Álvarez

Department of Informatics
King's College, London

jesus.dominguez_alvarez@kcl.ac.uk

2011

Originality avowal:

I, Jesus Dominguez Alvarez, verify that I am the sole author of this report, except where explicitly stated to the contrary. (25/4/2012)

Abstract

One of the major problems with software development in critical application systems is to verify the total correctness of the implementation. Part of the problem is that unexpected behavior may arise by modifications of the hardware, changes to the operative system or to the properties of the language. Creating applications that ensure run-time errors will not occur regardless of any set of specifications is one of the main benefits of certified software.

Certified software formally defines the syntax and behavior of software within a logical framework, including properties of the code and machine instructions. This framework is able to reason about the encoded language and its particular specifications by applying the same meta-logic that implemented the framework, building machine-checkable proofs that can be easily tested by a proof-assistant. By testing the validity of the proofs in the logic, we ensure that the software system produces programs that are correct-by-construction.

This paper aims to implement a toy imperative language, called SIMP, in the logical framework LF and to prove by means of the Twelf proof assistant that the semantics of the implemented language behave correctly with respect to the specifications given by the original definition of SIMP. This correctness will be tested by executing, in the framework, an algorithm that outputs any Fibonacci number, comparing results with the well-known sequence, as well as building proofs of the validity in the logic, and thereby, the correctness of the semantics of SIMP.

Acknowledgements:

I like to thank, first and foremost, Maribel Fernández for the time and energy spent on supervising my work. Her calmness and contagious enthusiasm helped me believe in myself in those times of darkness.

To Garrett, thank you for your moral support during all these years of study. I could not have managed without you at my side.

To my parents, for everything I am is thank to them.

And to my friends, that stoically listened to my moaning for four long years.

Contents

List of Tables

1 The Need for Certified Software

One of the long-standing ideals of Computer Science has been the development of correct software[1].

Our knowledge on how to describe, implement and understand the behavior of programs has improved exponentially, and this understanding is now been used during the software development life cycle in order to test and manually verify the correctness of moderately-sized programs in critical applications.

This has led to the need for technology capable of ensuring total correctness of critical application systems with respect to customer requirements, where errors are detected automatically even before being tested, thereby creating programs which are *correct-by-construction*[2].

Hall and Chapman[2] explain it by using the following scenario: When we buy a car, we expect the vehicle to be safe to drive, able to travel at a speed advertised by the manufacturer as well as easy to use by any regular driver. When we purchase a piece of software, we hold the same expectations towards its behavior: It will execute as declared by the provider. However, conventional software does not provide this type of confidence, behaving in ways completely unexpected to those implemented by the programmer. When the software is safety-critical, this uncertainty is unacceptable.

Software Reliability has been greatly improved by the use of type-safe languages like *Java*, *ML* or $\mathcal{C}\sharp$, guaranteeing that some run-time errors will not occur[4]. However, significant amount of code is still implemented using *unsafe* languages like $\mathcal{C}$++ because of the low abstraction level and architecture dependencies.

Leading research on *certified software* aims to formally define the syntax and behavior of software, including machine code, in order to check automatically for correctness by the inclusion of proofs that can be easily tested on *proof assistants*. Shao describes it as follows [5]:

> Certified software consists of a machine-executable program C plus a rigorous formal proof P (checkable by computer) that the software is free of bugs with respect to a particular dependability claim S. Both the proof P and the specification S are written using a general-purpose mathematical logic, this logic is also a programming language.

Dependability claims may include safety, security or other interesting properties of the code.

1.1 Components of a certified framework

There are several key elements required to build certified software, as described by Feng [4] and Shao [5]:

- *A formal model of the hardware.* Providing the operational semantics of all the machine instructions in order to trust the correctness of hardware implementation.

- *A mechanised meta-logic.* Expressive enough to allow programmers to describe all aspects of their programs. Mechanised so that formal proofs and specifications written as part of the software can be automatically checked by a computer.

- *A formal dependability claim for the software.* Including those specifications that describe properties of the code, such as memory safety, type safety, partial correctness, termination properties, resources usages or real-time properties.

- *A proof assistant.* To check the validity of all proofs following the inference rules of the meta-logic. It takes as input the certified package and the specifications, and it outputs whether the proof is valid in the logic.

Due to the complexity of building such a framework, this thesis will focus on mechanising the meta-theory of the imperative language $SIMP$[26], as explained on section 4, encoding its syntax and formalising its semantics. This mechanisation leads to a formal representation of well-typed terms and type-preserving operational semantics which offers the benefit of ensuring compatibility among compilers as well as enabling rigorous analysis of its properties. This analysis will also be demonstrated by the use of a proof assistant and a set of testing examples.

By designing the grammar of SIMP in a logical framework, this paper aims to build an equivalent logic system that can be used to reason about the correctness of the original specification.

By implementing this grammar in a proof assistant, we create a mechanised model able to build formal proofs of this correctness. This ensures that if the encoded system has a formal proof of its validity with respect to its specification, every program built in the original language is also correct, so much so, that only valid programs can be instantiating within this framework.

We aim to prove the correctness of our encoding by means of testing examples, however, this is not a formal proof of the equivalence of both deductive systems. Such formal proof is out of the scope of this paper.

For the purpose of the aforementioned mechanisation, the following section will first define the meaning of a *logical framework*, before surveying the various different logical frameworks available to choose between when defining our deductive system.

2 Logical Frameworks

Basin et al[7] explain logical frameworks as follows:

> A logical framework is a logic with an associated methodology that is employed for representing and using other logic, theories, and, more generally, formal systems. The emphasis is on reasoning *in* a logic, in the sense of simulating its derivations in the framework.

Therefore a logical framework is a meta-language able to define and manipulate syntactic structures such as types, proofs, programs and formulae. The idea of a logical framework as a language to describe and reason about deductive and type systems arises from *constructive logic*, where a proof of a proposition β consists on evidence for such β. Curry [10] and Howard [11] note that there is a computational logic to the process. For instance, to construct a proof of $A \to B$ (*introduction rule*), we need a proof of A and a proof of B ($true(A) \to true(B) \to true(A \to B)$). And vice versa, if we are given a proof of $A \to B$ and a proof of A (*elimination rule*), we can obtain a proof of B ($true(A \to B) \to true(A) \to true(B)$).

This leads to the *Curry-Howard correspondence*, also called *propositions-as-types*, that lays out the strong relationship between natural deduction in constructive logic and the *typed* λ*-calculus* where a type constructor also comes with two kinds of typing rules[12]:

1. an *introduction rule* describing the construction of elements of a type: *Type-abstraction.*

2. an *elimination rule* describing how elements of a type can be used: *Type-application.*

An opportunity for computation arises when an introduction term is immediately followed by another term as part of an elimination rule: this is called a *redex*. This observation gives rise to the following correspondence, clearly depicted in [12]:

LOGIC	PROGRAMMING LANGUAGES
propositions	types
proposition $P \supset Q$	type $P \to Q$
proposition $P \wedge Q$	type $P \times Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)

The table suggests that in order to show a judgement holds, we must show its premises also hold –thus to prove the validity of a judgement we only need to find terms to instantiate the proofs of the propositions that infer it, thereby creating a type signature and reducing proof-checking to type-checking.

A framework implementation is therefore able to check the validity of its derivations with respect to its given specification, this is given by the methodology of logical computation. Furthermore, as a logical framework is also a specification language [16], it should have a simple and uniform design, able to provide concise means to express the concepts of the intended application domain, so that meaningless expressions are detected statically.

The next sections describe and compare some of the available logical frameworks, ordered chronologically, and gives an understanding of their uses as well as describing the development of the field.

2.1 AUTOMATH

Historically the first logical framework[1] was *AUTOMATH*[16], conceived in 1967. Its goal was to formalise mathematics by means of expressing its underlying logic and thus verifying correctness of mathematical theories by means of an automated theorem prover.

Many of the ideas included in the system are still widely used on the methodology of the frameworks created afterwards, as De Bruijn reflects in [14], such as the use of a *typed λ-calculus* to define the system without linking it to any concrete logical language[2][18], the notion of *definitional equality* where two expressions are deemed definitionally equal if they both can be reduced to one and the same expression, the notion of *context* where a sequence of terms are declared previous to be used in the system. As well as being the first system exploiting the propositions-as-types described previously in section 2.

The rules used for describing mathematics in AUTOMATH are expressed in typed λ-calculus. A text consists of a sequence of lines, which together form a *Book* [15]. Lines consists of four parts: an *indicator*, an *identifier*, a *definition* and a *category*.

Each line introduces and declares a new name, this declaration can be defined in terms of names previously declared or as a new variable or a constant. This is called a *definition* of the name.

Names correspond to mathematical objects, to variables, to assertions, axioms, assumptions and theorems.

Next to the name there is an indication denoting its category, which it also has been previously introduced or corresponds to the symbol *type*, meaning it has been defined as a category to be used later by other names. Categories are unique.

We look now at a brief example of an AUTOMATH book:

[1] However, logical frameworks are based on logic and type theory, AUTOMATH is based on the formalisation of the notion of functions. We intentionally define it as such for being a pioneer in the field[18].

[2] Although the main difference among AUTOMATH systems was the type of λ-calculi included

line	indicator	identifier	definition	category
1	0	propos	PN	type
2	0	A	−	propos
3	A	B	−	propos
4	A,B	and	PN	propos
5	A	pf	PN	type
6	A,B	trueA	−	pf(A)
7	A,B,trueA	trueB	−	pf(B)
8	A,B,trueA,trueB	and_I	PN	pf(and)

- In line:

  1. We define a new primitive notion (basically a constant) `propos` and declare it a new type.
  2. `A` is defined as a new variable (variables are identified by −) of type `propos`.
  3. `B` is defined with respect to `A:propos`.
  4. Given propositions `A` and `B`, a new constant `and` is declared of type `propos` taking as parameters two elements of type `propos`: `A` and `B`.
  5. There is a new constant, this time we declare `pf` as a type that takes A as a parameter to construct proofs of propositions.
  6. We assume we have a proof of `A` from the context given by the propositions `A` and `B`.
  7. Similarly, given `A`, `B` and the predicate true(`A`), we assume true(`B`).
  8. We conclude by stating that if we have a proof of `A` and a proof of `B`, `and(A,B)` holds.

2.2 Hoare Logic

A different approach is described by C. Hoare in [19]. While in AUTOMATH we validate a theorem by constructing proofs of its premises in a typed language, now the focus is on deriving program correctness by means of axiomatic semantics(i.e, define the programming language semantics as a proof system).

The notion is that the finality of a program is to make changes on a given state, both connected by means of variables.

To prove the correctness of the program's functionality we make assertions[3] about the expected properties of the variables and the relationship among them when the program executes. These properties are written in mathematical logic and denoted as *post-condition*.

[3]invariants at specific programming points

The validity of the outcome of a program is closely related to the values given to the state before execution, this is the *precondition*.

The notation to show the relation between the different formulae is given as:

$$\{P\}\ C\ \{R\}$$

Listed below are the language-specific rules that construct the Hoare logic proof system. As they are based on an imperative language there should not be any difficulty on understanding the semantics, also, the example that follows should clarify any doubt. However, for a full explanation of each rule, the reader should go to [19, 20].

$$\frac{}{\vdash \{P\}skip\{P\}}(skip)\quad \frac{}{\vdash \{P[x \to E]\}\ x := E\ \{P\}}(assignment)$$

$$\frac{\vdash P' \Rightarrow P \quad \vdash \{P\}C\{R\}}{\vdash \{P'\}C\{R\}}(precondition\ strengthening)$$

$$\frac{\vdash \{P\}C\{R\} \quad \vdash R \Rightarrow R'}{\vdash \{P\}C\{R'\}}(post-condition\ weakening)$$

$$\frac{\vdash \{P\}C_1\{R\} \quad \vdash \{R\}C_2\{Q\}}{\vdash \{P\}C_1; C_2\{Q\}}(sequencing)\quad \frac{\vdash \{P \wedge E\}C\{P\}}{\vdash \{P\}\text{while } E \text{ do } C\{P \wedge \neg E\}}(while)$$

$$\frac{\vdash \{P \wedge E\}C_1\{R\} \quad \vdash \{P \wedge \neg E\}C_2\{R\}}{\vdash \{P\}\text{if } (E) \text{ then } \mathrm{C}_1 \text{ else } \mathrm{C}_2\{R\}}(if)$$

We now apply the logic rules to a sequence of given axioms example in order to prove the correctness of the following program taken from [20]:

```
{true}
R:= X;
Q:= 0;
while Y≤R
    do(R:= R-Y; Q:= Q+1)
{(X= R+(Y×Q)) ∧ R<Y}
```

By means of a *forward* proof we aim to prove first the properties of the given components (the first four assignment axioms) to put them together and prove the property of the program:

(1) assignment axiom $\{X= X+(Y\times 0)\}$ R:= X $\{X= R+(Y\times 0)\}$
(2) assignment axiom $\{X= R+(Y\times 0)\}$ Q:= 0 $\{X= R+(Y\times Q)\}$
(3) assignment axiom $\{X= (R-Y)+(Y\times(Q+1))\}$ R:= R-Y $\{X= R+(Y\times(Q+1))\}$
(4) assignment axiom $\{X= R+(Y\times(Q+1))\}$ Q:= Q+1 $\{X= R+(Y\times Q)\}$
(5) Applying sequencing-rule on (1) and (2):
$\{X= X+(Y\times 0)\}$ R:= X; Q:= 0 $\{X= R+(Y\times Q)\}$
(6) Applying sequencing-rule on (3) and (4):
$\{X= (R-Y)+(Y\times(Q+1))\}$ R:= R-Y; Q:= Q+1 $\{X= R+(Y\times Q)\}$
(7) By using a trivial arithmetic theorem we can derive:
$(X= R+(Y\times Q)) \wedge Y\leq R \Rightarrow (X= (R-Y)+(Y\times(Q+1)))$
(8) Hence we are now able to apply the precondition strengthening to (6) and (7):
$\{ (X= R+(Y\times Q)) \wedge Y\leq R\}$ R:= R-Y; Q:= Q+1 $\{X= R+(Y\times Q)\}$
(9) By looking at the precondition on (8) we decide to apply the while-rule:
$\{(X= R+(Y\times Q))\}$ while $Y\leq R$ do(R:= R-Y; Q:= Q+1) $\{(X= R+(Y\times Q)) \wedge \neg Y\leq R\}$
(10) Next we can glue the post-condition on (3) and the precondition on (9)
by using the sequencing-rule:
$\{X= X+(Y\times 0)\}$ R:= X; Q:= 0; while $Y\leq R$ do(R:= R-Y; Q:= Q+1)
$\{(X= R+(Y\times Q)) \wedge \neg Y\leq R\}$
(11) We know that $X= X+(Y\times 0) \equiv X=X$ and therefore true
(12) And also that $\neg Y\leq R \Rightarrow R<Y$
(13) By using preconditioning strengthening on (10)-(11)
and post-condition weakening on (12)-(10) we arrive to the program
we were aiming to prove.

One of the main drawbacks of using Hoare logic is that the proofs are hard to understand are hard to understand and debug due to the heavy use of encoding and repetition, as shown on the example.

Other problems stated in [21] are the difficulty to specify and prove properties about the heap (i.e. dynamically allocated, shared mutable objects), to specify recursive predicates, because they raise both theoretical and practical concerns. To be useful to functions, it would also need to specify which things do not change.

2.3 LCF and ML

The *LCF* (Logic for Computable Functions) proof system was one of the first interactive theorem provers, developed by Robin Milner in 1972.

Building from the AUTOMATH systems of typed λ-calculus and the *constructive type theories* of his contemporaries Martin-Lof and Dana Scott, it develops further by adding polymorphism, higher-order logic and , most importantly, by constructing an abstract data type whose predefined values are instances of axioms and whose operations are inference rules. This ensures that theorems could only be created by proof and thus type-checking would only allow values created from the axioms by applying a sequence of

inference rules.

To this end the functional programming language *ML* was created as a tool for constructing proofs on LCF, incorporating all the new features.

One of the first advantages of using abstract data types and ML type-checking is that proofs, generally done backwards from goals to axioms, could now be constructed going forwards, from axioms to goals, by means of the inference rules[4]. This lead to the creation of proof-search algorithms still widely used in current proof assistants like *Isabelle*[5], *HOL*[6] and *Coq*[7]. They are called *tactics* and *tacticals*.

Bertot and Castéran [23] give a definition on tactics and tacticals in their book about the proof assistant Coq:

> The user first states the formulae that need to be proved, these are the *goals*. The user then applies a series of commands to decompose these goals into sub-goals and so on and so forth, until each sub-goal generated is solved, constructing a proof of the goals from the solutions of the sub-goals. This can lead to very long proof scripts.
> Tacticals are operators that take a number of tactics as arguments returning new tactics. Using tacticals makes interactive proofs much more concise and readable.

Similarities between the ML framework and the LF framework are greater than its differences, as we show on section 2.4. However, these differences are the basis of our decision to implement the imperative language SIMP on LF.

It is not therefore necessary to describe the language paradigm of ML, or to provide an example, as done with previous frameworks (see 2.1 and 2.2). This would only add unnecessary length and repetition to our work.

A more constructive approach would be to present the description of the LF paradigm while particularly comparing those notions that differ in both frameworks.

2.4 The Logical Framework LF

Edinburgh LF or LF was designed in 1993 by Harper, Honsell and Plotkin [9] as a meta-language for representing deductive systems.

LF was also influenced by AUTOMATH systems and the constructive type theories of the 70s. However, the framework remains more closely related to the AUTOMATH language than ML. It provides an specification

[4] a methodology we described in the example on section 2.2
[5] http://www.cl.cam.ac.uk/research/hvg/isabelle/index.html
[6] http://www.cl.cam.ac.uk/research/hvg/HOL/
[7] http://coq.inria.fr/

language based on a general theory of logic, making the language logic-independent and with enough expressive power to specify a deductive system, its axioms, rules and proofs schematically [8].

These properties differ from Hoare logic, where the focus is on creating proofs about the behavior of a single program by checking the correctness of its state at each step of its execution.

Furthermore, this abstraction allows concise implementations of formulations like side conditions and variable occurrence at specification level, leading to the design of inference systems such as type systems, operational semantics, compilers, abstract machines and natural deduction, to name a few [8]. Its logic can also be used to reason about the properties of such systems, as a meta-logic.

Even though ML features polymorphism, this adds additional difficulty and even undecidability of type inference [22]. A consequence is the large amounts of type information we must give together with the program.

Similarly to ML, the type theory of LF is an extension of the typed λ-calculus. It derives its expressive power from the application of high-order representation techniques and dependent types, a feature not implemented in ML.

Dependently typed logic is a very powerful tool, able to define precisely the meaning of each formulation, or even including references in side type declarations. A prototypical example is the declaration of a function that returns the first element of a vector: $first :: \Pi n : nat.Vector(n+1) \rightarrow data$. This function takes as arguments a natural number n, a type Vector of size (n+1) and returns an element of type data. By using a dependent type to relate the connection between the arguments, the function not only returns the first element, but it also verifies statically that there are not out-of-bounds accesses allowed.

With an ordinary representation we would have added an additional mechanism to check that the function first is not applied to an empty array.

Then by using dependent types we can prove correctness of properties in types while writing certifying systems where proofs are already contained as part of the declarations. It makes LF a more powerful[8] and declarative[9] framework than ML.

We conclude by arguing that the Edinburgh LF is an ideal candidate for reasoning within an inference system as well as for reasoning about the properties of the system [8]. Providing a uniform and reliable alternative to the AUTOMATH (2.1), Hoare (2.2) and ML (2.3) frameworks, where a proof environment is implemented by means of a specific logical system, making the procedure both complex and time-consuming [17].

The following section describes the LF paradigm along with a specifi-

[8] in terms of static type checking.

[9] and thus, easier to understand

cation of how the meta-logic is applied in order to build a formal model of a given language. This model resultant is not yet mechanised, to achieve such goal we must use a proof assistant, therefore we finish the section by giving a description of the proof assistant *Twelf* as well as the reasons why we chose it over other proof assistants available.

With this knowledge we are then ready to formalise the specification and design of our project.

3 LF Background Specification

As we have briefly explained in section 1.1, our project aims at producing an encoding in LF of a simplified version of an imperative language, named SIMP, described in section 4.1.

Our aim is to develop the practise of formally defining the type safety and memory safety of imperative programming languages using the LF framework capabilities. By doing such formalisation we develop a realistic integration of the language so much so that its abstract syntax could be generated by a parser, and its semantics could be applied to the abstract syntax representation by means of specifying them in an algorithmic fashion.

To conclude, we will use our mechanised formal encoding of SIMP to reason about a given program written in this language, particularly to create proofs about its behaviour. This approach has already being seen previously when discussing Hoare logic (2.2), where we reason about each step of the program's execution.

We argued about the length and complexity of the proofs it develops, we now show a different approach, by using an interactive proof assistant to build the desired proofs.

In order to realise the extent of our project, we continue by providing an specification of how LF operates, a definition of its structure and the elements that are part of this structure. Examples are written in order to give a practical understanding of the theory of LF. Also we lay the foundations of *adequacy*, and explain the importance of our model to be adequate with respect to the language SIMP.

3.1 Encoding Syntax and Semantics

The specification of a type system proceeds in two stages. first we define the syntax and then its derivations via axioms and rules of inference.

3.1.1 Representing Syntax

The LF calculus is a three-level calculus [24] for Objects, Families and Kinds. Families are classified by Kinds, Objects are classified by Types, that is, Families of Kind Type.

$$
\begin{array}{llll}
\textit{Kinds} & K & ::= & \mathit{Type} \mid \Pi x{:}A.K \\
\textit{Families} & A, B & ::= & a \mid \Pi x{:}A.B \mid \lambda x{:}A.B \mid AM \\
\textit{Objects} & M, N & ::= & c \mid x \mid \lambda x{:}A.M \mid MN
\end{array}
$$

We use a for constants at the level of families and c for constants at the level of objects.

We use *signatures* to assign Kinds and Types to constants by means of declarations. We use *contexts* to assign Types to variables.

The formalisation of a type system like SIMP starts by taking the informal representation of the language and correctly represent it in the LF framework. It is done by presenting the syntax and derivations of the object language as canonical forms[10] of associated LF types, denoting a *compositional bijection* between the informal representation of the SIMP and its related canonical form. This ensures that any reasoning in or about LF applies also to the informal definition and vice versa; we say that the LF representation of the language is *adequate*.

This adequacy is established by a function that inductively maps canonical terms and variables in LF and vice versa.
Let's recapitulate by using an example:

Our language must represent primitive operations to manipulate data types such as numbers. Therefore a syntactic category of SIMP is that of the natural numbers, denoted in LF by writing $nat :: \mathit{Type}$.

In order to represent the family of primitive operations of Type nat we declare a signature Σ_{nat} that includes a constant for each term constructor that deals with arithmetical expressions.

$$
\begin{array}{lcl}
\mathit{zero} & : & \mathit{nat} \\
\mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{plus} & : & \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\
\mathit{times} & : & \mathit{nat} \to \mathit{nat} \to \mathit{nat}
\end{array}
$$

and a context $\Gamma_X : x_1{:}nat, \ldots, x_n{:}nat$ where x is a finite set of variables.

We encode the terms of the object language in LF by a function $\lceil . \rceil$ mapping terms of arithmetic expressions(with free variables in X) to terms of type nat in Σ_{nat} and Γ_X. This encoding is defined by induction on the structure of terms as follows:

$$
\begin{array}{rcl}
\lceil x \rceil_X & = & x \\
\lceil 0 \rceil_X & = & \mathit{zero} \\
\lceil \mathit{succ}(x) \rceil_X & = & \mathit{succ}\ \lceil x \rceil_X \\
\lceil x_1 + x_2 \rceil_X & = & \mathit{plus}\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X \\
\lceil x_1 \times x_2 \rceil_X & = & \mathit{times}\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X
\end{array}
$$

[10] essentially long $\beta\eta$-normal forms

3.1.2 Representing Semantics

The next step towards formalisation is the representation of the semantics of the given system.

Having established an LF constant declaration for each syntactic constructor in the type system, we now proceed to represent proofs in LF. These proofs are constructed by means of the axioms and inference rules that form the semantics of the language.

The strategy is based on the Curry-Howard isomorphism explained in section 2, denominated here *judgement as types and derivations as objects*[9].

The approach is similar to the declaration of syntactic constructors in 3.1.1, where a formula is adequate if we can successfully derive its canonical form in LF by means of the function $\lceil . \rceil$.

Hence, by following the example previously given in 2, to define a judgement that states a formula A *is true*, we declare a type family *proof* indexed by representations of formulae[11] used as proof constructors.

Then $proof\lceil A \rceil$ is the representation of A *is true* and by applying the *adequacy theorem*[9] we conclude that $proof\lceil A \rceil$ is valid in the meta-logic iff we are able to derive A *is true*. Basically, we are proving the validity of formulae in our deductive system by inhabiting the types of the constructors in LF with object terms representing the semantics of the type system.
For instance:

We want to prove that $A \wedge B$ is true for any given A and B propositions where A and B are true.

We declare a Type $prop :: Type$ for propositions and a Kind $proof : prop \rightarrow Type$ to represent valid proofs of propositions.

First we translate the formula to LF by applying the function $\lceil . \rceil$:

$$\lceil A \wedge B \rceil_X = and\ \lceil A \rceil_X\ \lceil B \rceil_X$$

We add to the signature the constant *and* as $and : prop \rightarrow prop \rightarrow prop$ declaring that to construct the family type *and* we need to inhabit it with two propositions, A and B, forming the prop Type $andAB$. From natural deduction we know that to proof $A \wedge B$ we need a proof of A and a proof of B (conjunction introduction), thereby we show it as part of our translation from natural deduction to LF:

$$\left[\begin{array}{cc} D_A & D_B \\ \vdots & \vdots \\ A & B \\ \hline A \wedge B & \end{array}\right]_X = and_{in}\ \lceil A \rceil_X\ \lceil B \rceil_X\ \lceil D_A \rceil_X\ \lceil D_B \rceil_X$$

where D are sub-derivations that leads to a proof of A *is true* and B *is true*. Hence our proof is defined by:

[11] these formulae are the axioms and inference rules of the semantics of SIMP

$$and_{in} : \Pi A{:}prop.\Pi B{:}prop.\ proof(A) \rightarrow proof(B) \rightarrow proof(and(AB))$$

adequately indicating that if we have two propositions A and B, and proofs that A *is true* and B *is true*, we can successfully derive the judgement $A \wedge B$ as also true.

Pierce explains in [25] that the main advantage of using LF for proof representation is that proof validity can be checked by a simple type-checking algorithm(described in [9]) using the LF typing rules to verify that an object M has type proof$\lceil F \rceil$ in the context of the signature Σ declaring the valid proof rules.

3.2 Twelf Proof Assistant

The Twelf proof assistant [28] is an implementation of the LF logical framework used as a vehicle for writing machine-checked proofs of type system theorems[27].

Frank Pfenning and Carsten Schürmann [29], main authors of the software, describe it as follows:

Twelf is a meta-logical framework for the specication, implementation, and meta-theory of deductive systems from the theory of programming languages and logics. It relies on the LF type theory and the judgements-as-types methodology for specication , a constraint logic programming interpreter for implementation, and the meta-logic M_2 for reasoning about object languages encoded in LF. It is a signicant extension and complete reimplementation of the Elf system.

The syntax and behavior of a language formalised in LF can be checked by the Twelf proof assistant, along with theorems about the properties and requirements of such language. This results in a consistent and reliable proof of the language and its capabilities that can be used to ensure correctness of its original specification, thereby creating programs that become correct by construction. Furthermore, it guarantees the preservation of the grammar if extending the language by supporting the modifications with a formal machine-checkable proof.

Another reason for the use of Twelf in our project is that Twelf has the ability to directly express proofs in the formal language LF, making it more attractive than Isabelle or Coq where a complete proof consists of a list of tacticals but not the proof itself.

As LF is a meta-logic mechanised within Twelf, the definition of the formal model of SIMP is defined precisely with the syntax of Twelf, which, in any case, is equivalent to that of LF. As a result, in our thesis both design and implementation are unified as a single stage, sharing the syntax and set of rules.

The encoding in Twelf represents a formal implementation of the informal toy language SIMP, described in section 4.1, where each of the signatures

in Twelf has a counterpart in SIMP, unless stated otherwise. However, the adequacy of the encoding with respect to SIMP is only informally verified.

The methodology of the proof assistant is explained along with the implementation of the language, generating a Twelf user guide while building our formal model.

4 Specification

The aim of the project is to build a mechanised meta-logic of the imperative language SIMP. This logic is one of the key elements required to build certified software as we have already depicted in section 1.1.

The formal model we are about to design is a realistic representation of SIMP such that its abstract syntax and operational semantics could be generated by a parser allowing representations of programs to be run on our formalisation. We do not aim to build such parser yet Twelf has a built-in server in which queries and proofs can be written and resolved by means of the meta-logic specified. We make use of this feature in our project to write and test programs written in our formalised grammar. This is explained in more detail further within this section.

One of the key areas of the project is to build an isomorphic representation of SIMP in Twelf, so that each syntactic term has a unique related term in our formal model and vice versa, preserving each of the features and functionality SIMP has, as well as establishing type safety and building a formal model of the memory with identical capabilities to that of the model described in SIMP while ensuring its correct manipulation. This ensures that any reasoning in or about the mechanised model is also valid in the original representation (as explained in section 3.1.1). Nevertheless, a formal proof of equivalence between our model and the original model is out of the scope of this thesis.

The mechanisation of our language provides a precise specification for programmers, ensures compatibility among compilers and enables rigorous analysis of its properties[6]. These properties define the correctness of the language, some can be as simple as the associative property of a particular arithmetic operation, for instance; whereas others may need longer and more complex theorems together with associated lemmas, that must have been previously proved, to successfully complete the proof of one of these theorems.

It is not difficult to see that writing these proofs by hand becomes a long and tedious work, where any minimal change made to the language leads to a complete remake of any of its proofs. The mechanisation of our language in Twelf overcomes these obstacles. Verification of the language in LF is done by writing theorems with the same grammar used to encode the programming language while employing the proof assistant to check on

the correctness of such meta-theory and therefore, establishing adequate LF representations of the original specification as well.

However, this project leaves the definition of meta-theorems and their proofs for a further study and focuses its goal on a correct encoding of the language SIMP, capable of produce valid results. We reason about the behavior of the language by creating proofs about the behavior of individuals programs, in a style similar to Hoare logic, seen in section 2.2.

Lacking a formal proof of adequacy, we make extensive use of testing in order to provide a means of verifying the correctness of our implementation as a valid imperative language. This entails testing separately the different syntactic structures that form the encoding, to conclude by building a complex program that applies the full capabilities of the semantics. This program is based on the Fibonacci algorithm, using its well known sequence of numbers to validate our results.

To build an encoding of SIMP in Twelf we require:

- the imperative language SIMP, illustrated in section 4.1,
- the logical framework LF, described in section 3,
- Twelf's proof assistant, briefly described in section 3.2, and in more detail later in section 5,
- and the editor *Emacs*[12], as the Twelf server can be installed as an extension, enabling the editing and testing of the formal model at hand.

4.1 The Imperative Programming Language SIMP

4.1.1 Abstract Syntax

Programs

$$P ::= C|E|B$$

Commands

$$C ::= skip \mid l := E \mid C; C \mid if\ B\ then\ C\ else\ C \mid while\ B\ do\ C$$

Integer Expressions

$$E ::= !l \mid n \mid E\ op\ E$$
$$op ::= + \mid - \mid \times \mid \div$$

Boolean Expressions

[12] http://www.gnu.org/software/emacs/

$$B ::= \ True \mid False \mid E \ bop \ E \mid \neg B \mid B \ \wedge \ B$$
$$bop ::= < \ \mid \ > \ \mid \ =$$

where

- $n \in Z$
- $B = \{True, \ False\}$
- $l \in L = \{l_0, l_1, ...\}$(locations or variables)
- The expression $!l$ denotes the value stored in l.

4.1.2 Structural Operational Semantics

The *Reduction* Relation. *Small-step Semantics* .
For Expressions:

$$\frac{}{\langle !l, s\rangle \rightarrow \langle n, s\rangle, \text{ if } s(l) = n}(var)$$

$$\frac{}{\langle n_1 \text{ op } n_2, s\rangle \rightarrow \langle n, s\rangle, \text{ if } n = (n_1 \text{ op } n_2)}(op)$$

$$\frac{}{\langle n_1 \text{ bop } n_2, s\rangle \rightarrow \langle b, s\rangle, \text{ if } b = (n_1 \text{ bop } n_2)}(bop)$$

$$\frac{\langle E_1, s\rangle \rightarrow \langle E_1', s'\rangle}{\langle E_1 \text{ op } E_2, s\rangle \rightarrow \langle E_1' \text{ op } E_2, s'\rangle}(op_L) \ \frac{\langle E_2, s\rangle \rightarrow \langle E_2', s'\rangle}{\langle E_1 \text{ op } E_2, s\rangle \rightarrow \langle E_1 \text{ op } E_2,' s'\rangle}(op_R)$$

$$\frac{\langle E_1, s\rangle \rightarrow \langle E_1', s'\rangle}{\langle E_1 \text{ bop } E_2, s\rangle \rightarrow \langle E_1' \text{ bop } E_2, s'\rangle}(bop_L) \ \frac{\langle E_2, s\rangle \rightarrow \langle E_2', s'\rangle}{\langle E_1 \text{ bop } E_2, s\rangle \rightarrow \langle E_1 \text{ bop } E_2,' s'\rangle}(bop_R)$$

$$\frac{}{\langle b_1 \wedge b_2, s\rangle \rightarrow \langle b, s\rangle, \text{ if } b = (b_1 \text{ and } b_2)}(and)$$

$$\frac{}{\langle \neg b, s\rangle \rightarrow \langle b', s\rangle, \text{ if } b' = \text{not } b}(not) \ \frac{\langle B_1, s\rangle \rightarrow \langle B_1', s'\rangle}{\langle \neg B_1, s\rangle \rightarrow \langle \neg B_1', s'\rangle}(notArg)$$

$$\frac{\langle B_1, s\rangle \rightarrow \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \rightarrow \langle B_1' \wedge B_2, s'\rangle}(and_L) \ \frac{\langle B_2, s\rangle \rightarrow \langle B_2', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \rightarrow \langle B_1 \wedge B_2', s'\rangle}(and_R)$$

For Commands:

$$\frac{\langle E, s\rangle \to \langle E', s'\rangle}{\langle l := E, s\rangle \to \langle l := E', s'\rangle}(:=_R) \quad \frac{}{\langle l := n, s\rangle \to \langle skip, s[l \mapsto n']\rangle}(:=)$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}(seq) \quad \frac{}{\langle skip; C, s\rangle \to \langle C, s\rangle}(skip)$$

$$\frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \to \langle if\ B'\ then\ C_1\ else\ C_2, s'\rangle}(if)$$

$$\frac{}{\langle if\ True\ then\ C_1\ else\ C_2, s\rangle \to \langle C_1, s\rangle}(if_\top)$$

$$\frac{}{\langle if\ False\ then\ C_1\ else\ C_2, s\rangle \to \langle C_2, s\rangle}(if_\bot)$$

$$\frac{}{\langle while\ B\ do\ C, s\rangle \to \langle if\ B\ then\ (C; while\ B\ do\ C)\ else\ skip, s\rangle}(while)$$

The *Evaluation* Relation. *Big-step Semantics* . For Expressions:

$$\frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle,\ if\ c \in Z \cup \{True,\ False\}}(const)$$

$$\frac{}{\langle !l, s\rangle \Downarrow \langle n, s\rangle,\ if\ s(l) = n}(var)$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle \quad \langle B_2, s'\rangle \Downarrow \langle b_2, s''\rangle}{\langle B_1 \wedge B_2, s\rangle \Downarrow \langle b, s''\rangle,\ if\ b = b_1\ and\ b_2}(and)$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle}{\langle \neg B_1, s\rangle \Downarrow \langle b, s''\rangle,\ if\ b = not\ b_1}(not)$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1\ op\ E_2, s\rangle \Downarrow \langle n, s''\rangle,\ if\ n = n_1\ op\ n_2}(op)$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1\ bop\ E_2, s\rangle \Downarrow \langle b, s''\rangle,\ if\ b = n_1\ bop\ n_2}(bop)$$

For Commands

$$\frac{}{\langle skip, s\rangle \Downarrow \langle skip, s\rangle}(skip) \quad \frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle l := E, s\rangle \Downarrow \langle skip, s'[l \mapsto n]\rangle}(:=)$$

$$\frac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip, s''\rangle}(seq)$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}(if_\top)$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}(if_\bot)$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C, s'\rangle \Downarrow \langle skip, s''\rangle \quad \langle while\ B\ do\ C, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle while\ B\ do\ C, s\rangle \Downarrow \langle skip, s'''\rangle}(while_\top)$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle}{\langle while\ B\ do\ C, s\rangle \Downarrow \langle skip, s'\rangle}(while_\bot)$$

We have given a specification of the goals this project aims to achieve along with the tools necessary to do so. The next section focuses on the design and implementation of our model, providing a detailed definition of the syntax and dynamic semantics of SIMP in Twelf as well as a justification of the choices made.

5 Design

This section deals with the design of a formal model of SIMP in LF. The design is directly built as an implementation in Twelf, separated into two stages, the syntax and the semantics formalisation. We start with the syntax.

5.1 Encoding of Syntax

Encoding the abstract syntax of terms and types of SIMP involves firstly the application of the function $\lceil . \rceil$, explained in section 3 and illustrated in Table 5.1, on each of the terms that constitute the language.

In our encoding two syntactic levels with identical operators are differentiated with unique constants: the level of primitive data types and the level

of expressions. In SIMP, both syntactic levels are denominated similarly, however, if we look at their behavior in the semantic representation in section 4.1.2, we observe an internal distinction where data types are classified as canonical forms of expression types. This is also our aim when encoding the logic in Twelf.

Additionally, LF terms representing primitive types have an extra argument compared to their SIMP counterparts. This is due to explicitly defining the output as part of the signature, allowing expressions to directly operate on it.

$\lceil x \rceil_X$	=	x
$\lceil l \rceil_X$	=	$loc\ \lceil x \rceil$
primitive types:		
$\lceil true \rceil_X$	=	$true$
$\lceil false \rceil_X$	=	$false$
$\lceil 0 \rceil_X$	=	$zero$
$\lceil successor(x) \rceil_X$	=	$suc\ \lceil x \rceil_X$
$\lceil x_1 + x_2 \rceil_X$	=	$plus\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil x_1 - x_2 \rceil_X$	=	$minus\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil x_1 \div x_2 \rceil_X$	=	$div\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X\ \lceil x_4 \rceil_X$
$\lceil x_1 \times x_2 \rceil_X$	=	$product\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil x_1 \wedge x_2 \rceil_X$	=	$and\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil \neg x \rceil_X$	=	$negation\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X$
$\lceil x_1 = x_2 \rceil_X$	=	$equal\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil x_1 > x_2 \rceil_X$	=	$gt\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil x_1 < x_2 \rceil_X$	=	$lt\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$

expressions:		
$\lceil x_1 + x_2 \rceil_X$	=	$\lceil x_1 \rceil_X + \lceil x_2 \rceil_X$
$\lceil x_1 - x_2 \rceil_X$	=	$\lceil x_1 \rceil_X - \lceil x_2 \rceil_X$
$\lceil x_1 \times x_2 \rceil_X$	=	$\lceil x_1 \rceil_X * \lceil x_2 \rceil_X$
$\lceil x_1 \div x_2 \rceil_X$	=	$\lceil x_1 \rceil_X\ /\ \lceil x_2 \rceil_X$
$\lceil x_1 \wedge x_2 \rceil_X$	=	$\lceil x_1 \rceil_X \wedge \lceil x_2 \rceil_X$
$\lceil \neg x \rceil_X$	=	$not\ \lceil x \rceil_X$
$\lceil x_1 = x_2 \rceil_X$	=	$\lceil x_1 \rceil_X == \lceil x_2 \rceil_X$
$\lceil x_1 > x_2 \rceil_X$	=	$\lceil x_1 \rceil_X > \lceil x_2 \rceil_X$
$\lceil x_1 > x_2 \rceil_X$	=	$\lceil x_1 \rceil_X > \lceil x_2 \rceil_X$

commands:		
$\lceil if\ x_1\ then\ x_2\ else\ x_3 \rceil_X$	=	$if\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X\ \lceil x_3 \rceil_X$
$\lceil while\ x_1\ do\ x_2 \rceil_X$	=	$while\ \lceil x_1 \rceil_X\ \lceil x_2 \rceil_X$
$\lceil !\ x \rceil_X$	=	$!\ \lceil x \rceil_X$
$\lceil skip \rceil_X$	=	$skip$
$\lceil l := x \rceil_X$	=	$loc\ \lceil x_1 \rceil_X\ assigns\ \lceil x_2 \rceil_X$
$\lceil x_1\ ;\ x_2 \rceil_X$	=	$\lceil x_1 \rceil_X\ ;\ \lceil x_2 \rceil_X$

Table 5.1: Mapping terms in SIMP to terms in LF with function $\lceil . \rceil$

5.2 Encoding of Semantics

5.2.1 Base Types

Twelf uses the same syntax to define both Kinds and family type declarations: $name : Type$. Thus, a language to manipulate primitive data types in SIMP has the following Types and Kinds in LF encoding:

$nat : type.$	$bool : type.$
$zero : nat.$	$true : bool.$
$suc : nat \rightarrow nat.$	$false : bool.$

where *nat* represents the base type of the natural numbers and *bool* that of the Boolean. *type* is a constructor to declare types in Twelf but it cannot be used as a type itself. Every natural number is computed with respect

to the constant value zero by the function *suc* that takes a *nat* data type and produces its successor. This is similar to the representation of natural numbers in Lambda calculus. A function computing the predecessor of a number could have also been defined, however, it would not add any extra to our encoding and thus we find it unnecessary to implement. Both constants, $true$ and $false$, are self-explanatory.

Primitive base types and related terms depicted above are not yet expressions.This differentiation of expressions as separate data types is critical for commands such as $if\ E\ then\ E\ else\ E$ or $while\ E\ do\ E$ to recognise the different primitive types that the variable E entails, and behave accordingly. Having declared both natural numbers and Boolean only as expression types would have led to a distinct behavior of that specified by the semantics of SIMP.

Table 5.2 illustrates some of the results of the encoding and introduce us to a number of features of the Twelf proof assistant.

$nat : type.\ \%name\ nat\ N.$	$bool : type.\ \%name\ bool\ B$
$zero : nat.$ $suc : nat \rightarrow nat.\ \%prefix\ 8\ suc.$	$true : bool.$ $false : bool.$
$plus : nat \rightarrow nat \rightarrow nat \rightarrow type.$ $plus/zero : plus\ zero\ N\ N.$ $plus/suc : plus\ (suc\ N1)\ N2\ (suc\ N3)$ $\leftarrow plus\ N1\ N2\ N3.$	$and : bool \rightarrow bool \rightarrow bool \rightarrow type.$ $and/1\ \ and\ true\ B\ B.$ $and/2\ \ and\ B\ true\ B.$ $and/3\ \ and\ B\ B\ B.$
$product : nat \rightarrow nat \rightarrow nat \rightarrow type.$ $product/zero : product\ zero\ N\ zero.$ $product/suc : product\ (suc\ N1)\ N2\ (suc\ N3')$ $\leftarrow plus\ N2\ N3\ N3'$ $\leftarrow product\ N1\ N2\ N3.$	$gt : nat \rightarrow nat \rightarrow bool \rightarrow type.$ $gt/f : gt\ zero\ N\ false.$ $gt/t : gt\ (suc\ N)\ zero\ true.$ $gt/suc : gt\ (suc\ N1)\ N2\ B$ $\leftarrow gt\ N1\ N2\ B.$

$$\%abbrev \quad lt : nat \rightarrow nat \rightarrow bool \rightarrow type \ = \ [N1]\ [N2]\ [B]\ gt\ N2\ N1\ B.$$

Table 5.2: Encoding of primitive types

The keyword $\%\ name$ followed by the name of the type and a meta-variable helps Twelf to infer which type is expected when encountered with the chosen capital letter, so much so that an error message is displayed if, for instance, we write *suc B* on our logic after declaring B a meta-variable of type Boolean. Additionally, it helps the user to understand error messages, as Twelf specifies any error type by using our provided variable declarations instead of a generic variable for any of the possible types. The keyword $\%\ prefix$ and its attributes written beside the declaration of *suc* specify the location and precedence of the term over its parameters and other possible terms. In this case, *suc* is a prefix term thus taking a parameter placed after the term, with a precedence of 8 (where $0 \leq prec < 1000$). This numbering becomes particularly useful when declaring arithmetic operations

where some must be applied before others. It also helps Twelf to eliminate the need for parentheses where the keyword clarifies the structure of the term to be constructed.

The definition of the primitive type semantics demonstrate the general pattern of *term reconstruction* used by Twelf to validate judgements by means of instantiating inference rules. To describe this methodology we look at the declaration of addition in Table 5.2.

First, the constant *plus* is declared as a function that produces a *type*. This is the judgement. The type *plus* needs three arguments of type nat to be constructed: the first two arguments are treated as inputs and the last one is considered the output. Instantiating all three types with adequate terms creates an object term of type *plus*, this term is a proof of the validity of *plus*.

Having declared the type of constant *plus*, it follows by describing the inference rules that form its family type. These are written in a style similar to prolog, constructing a logic program whose behavior Twelf validates by using *structural induction* over the derivations in order to reconstruct the terms missing until arriving to a terminal state.
This implies that, for the logic program to terminate, we must build a base case of type *plus*. Similarly, to reconstruct the logic, a recursive call must be defined. Twelf uses the instantiation of the arguments with the correspondent parameters as a proof of validity of the judgement *plus* while producing the desired result of adding two natural numbers, expressed by the reconstruction of the output term.

Each derivation must be given a different name for Twelf to refer to it in the proofs. The convention is to write the name of the judgement and the case name together, separated by a slash.

Plus base case is defined as a number added to zero, producing the same number:

$$\frac{}{zero + N \Downarrow N} \qquad plus/zero : plus\ zero\ N\ N.$$

Capital N denotes meta-variables in the syntax of Twelf, parameters to instantiate types given by the user or reconstructed by the proof assistant.

The recursive call is based on proving, inductively over the structure of *plus*, that by assuming the sum of N1, N2 as valid , These assumptions are validated when unified with the base case.

$$\frac{N1 + N2}{(suc\ N1) + N2} \qquad plus/suc : plus\ (suc\ N1)\ N2\ (suc\ N3) \leftarrow plus\ N1\ N2\ N3.$$

Notice that the consequence is written before the antecedent, preceded by $\leftarrow$.
It could have been written conversely, however the backward style reflects Twelf's order of evaluation.
For instance, to solve the sum of numbers two and one, denoted by the

object term M = *plus* (*suc suc zero*) (*suc zero*) N, Twelf begins by trying to unify M with *plus*/*zero* but it fails as *zero* $\neq$ (*suc suc zero*). It continues down the next rule and attempts to unify M with *plus*/*suc*, succeeding by substituting

- $\{N1 \mapsto suc\ zero\}$,
- $\{N2 \mapsto suc\ zero\}$ and
- $\{N3 \mapsto suc\ N\}$.

It follows by making a recursive call to *plus* $N1$ $N2$ $N3$, failing again at unification with *plus*/*zero*, yet succeeding with *plus*/*suc* with substitutions[13]:

- $N1' = suc\ N1$ thus $\{N1 \mapsto zero\}$
- $N2' = N2$, then $\{N2 \mapsto suc\ zero\}$ and
- $suc\ N3' = suc\ N3$ resulting in $\{N3 \mapsto suc\ suc\ N3'\}$.

Another recursive call to *plus* $N1$ $N2$ $N3$, this time unifying with *plus*/*zero* with the following substitutions:

- *zero* = *zero*,
- $N2 = N =$ *suc zero* and
- $N = N3$, where $\{N3 \mapsto (suc\ suc\ N3')\}$, resulting in the solution to the initial query M = *plus* (*suc suc zero*) (*suc zero*) N with the reconstructed term $N =$ (*suc suc suc zero*).

In a nutshell, we describe Twelf's logic as a backtracking search and unification algorithm reminiscent of the unification logic yet more complex as it must deal with dependent types. This is expressed by means of *immediate implication*, where a goal G (any given query) represents a successful unification with a clause of type $H \leftarrow E_1 \leftarrow ... \leftarrow E_n$, possibly with meta-variables, under a substitution θ where each sub-goal θB_i has been previously proved.

Along with the solution, Twelf also provides the recursive calls that constitute the body of the logic program, these derivation are the *proof witness* of the judgement plus, in this case, the derivation *plus*/*suc* (*plus*/*suc* (*plus*/*zero*)) given with the parameters implicitly quantified. The explicit version is given by
plus/*suc* (*suc zero*) (*suc zero*) (*suc suc zero*)
(*plus*/*suc zero* (*suc zero*) (*suc zero*)
(*plus*/*zero* (*suc zero*))).
Twelf provides implicit quantifiers with solutions, yet it explicitly quantifies user's declarations when implemented. For instance, our definition of

[13]to avoid confusion we rename $N1$ $N2$ $N3$ as $N1'$ $N2'$ *and* $N3'$ respectively

plus/zero is illustrated by Twelf as $plus/zero : \{N : nat\}\ plus\ zero\ N\ N.$, expressing the dependency between the parameter and the derivation to be constructed. Any failing on inferring the type of parameters results in an error report.

The family type *product* shows a more complex representation, where calls to other types are necessary in order to proof the validate of the term, in this case calls to the type *plus*. It is based on the fact that any number N multiplied K times is equivalent to adding such number K times: $N \times K = M_1 + ... + M_K$. Hence, to prove $N \times K$, we must inductively prove $N \times (K-1)$ equivalent to proving $M_1 + ... + M_{K-1}$, and continuing until the base case of zero is reached.

Another method to declare derivations of a family type is to precisely defined the number of possible base cases that constitute the proof of the judgement, as seen on type *and* in table 5.2.
We could have declared a signature $proof$, following the example in section 2.4, and define constants of type $proof$ representing introduction and elimination rules of *and* and any other first-order logic term in SIMP. However, as Twelf is used as a specification language to encode logic of SIMP, the type term $proof(and\ A\ B)$ would not be an adequate solution for an evaluation of a Boolean operation in SIMP as it does not constitute a canonical value,and thus opting to define the encoding of *and* as a family type with an explicit output that returns a canonical form. The output is then returned as an argument that solves a query. Or to build up further, more complex expressions. This is explained in more detailed when encoding the dynamic semantics of the language in section 5.2.4.

Finally, the keyword $\%abbrev$ defines a constant that it is immediately replaced by the given definition on the right side of the = sign. In this case, switching the original position of the parameters in the function *gt* (*greater than*) to build the function *lt* (*less than*).

The remaining of the basic operations do not offer any additional difficulty and thus are left out of table 5.2. The complete list is illustrated in appendix A.

5.2.2 Expression Terms

After encoding the primitive data types and their operations, we continue by encoding natural numbers as integers, Boolean as propositions(*prop*) and both integers and propositions, along with commands, as expressions. Notice that every element in Table 5.3 belongs to the family type *exp* and no inference rules have been described. This is because SIMP treats expressions at a higher level than primitive types, and the evaluation of each expression depends on the behavior and the command applied to it. It functions as a common type to all syntactic terms of the language, in order to build sentences meaningful, by following the rules of the logic. This becomes

clearer when describing the semantics of the language in section 5.2.4.

$exp : type.\ \%name\ exp\ E.$	
$skip : exp.$ $loc : nat \to exp.$ $int : nat \to exp.$ $prop : bool \to exp.$	
$+ : exp \to exp \to exp.\ \%infix\ left\ 8\ +.$	$- : exp \to exp \to exp.\ \%infix\ left\ 8\ -.$
$* : exp \to exp \to exp.\ \%infix\ left\ 9\ *.$	$/ : exp \to exp \to exp.\ \%infix\ left\ 9\ /.$
$not : exp \to exp.\ \%prefix\ 7\ not.$	$\wedge : exp \to exp \to exp.\ \%infix\ left\ 6\ \wedge.$
$>: exp \to exp \to .\ \%infix\ left\ 7\ >.$	$<: exp \to exp \to exp.\ \%infix\ left\ 7\ <.$
$==: exp \to exp \to exp.\ \%infix\ none\ 6\ ==.$	
$if : exp \to exp \to exp \to exp.\ \%prefix\ 2\ if.$	$while : exp \to exp \to exp.\ \%prefix\ 2\ while.$
	$; : exp \to exp \to exp.\ \%infix\ left\ 1\ ;.$
$ref : exp \to exp.\ \%prefix\ 5\ ref.$	$! : exp \to exp.\ \%prefix\ 10\ !.$
$assigns : exp \to exp \to exp.\ \%infix\ none\ 4\ assigns.$	

Table 5.3: Encoding of expressions

The most relevant parts to comprehend at this point are, first, how memory is syntactically represented in our design, and second, how the preference values are going to affect our language.

There are four constant terms for memory access, loc, $ref, assigns$ and $!$:

- $loc : nat \to exp$ is an abstract type representing each memory cell in the heap. Each value nat expressed within this term uniquely indexes a memory space, and thus it is referenced in the program logic when accessing the memory. A primitive type nat could have been directly applied without need to represent a new kind, however, this represents a syntactic ambiguity where natural numbers and location variables are defined by the same type. Our language must have a means to clearly represent variables at a syntactic level, here we named them by explicitly indicating the location index in the heap with the term $loc\ N$, where N represents a unique location within the heap.

- $ref : exp \to exp$ allocates a new memory space for a given expression in the heap and returns the location index $loc\ N$. Therefore this command creates a fresh variable by storing an expression value in the heap and returning its index. This index is a location variable to be used throughout the rest of the program as a reference to the memory cell N.

- *assigns* basically takes two arguments, an expression to update the memory, and the location of the memory cell as given by a local variable.

- If *loc N* references the memory cell, ! : *exp* → *exp* directly references its content in order to be applied to the program; written as ! *loc N*.

We have defined each of the operators of the language as *left-associative*, or *none* if it should not associate with another command. The numbering declared beside decides which commands and operators must be evaluated first by having a higher value than the rest. We have decided to give preference to arithmetic operators and commands with just one parameter, thus if a command encounters a long sequence of operations, Twelf attempts to reduce as much as possible the operations before applying the command. Sometimes it encounters ambiguous formulae with nested commands that leads to unexpected evaluations. Also, a formula with a command dereferencing a local variable starts by reducing this variable to the value contained in the memory cell. To conclude, the expression *skip* is defined as a command in SIMP, however, we have decided to treat it as an expression in our encoding, nevertheless it retains the same functionality of its equivalent. The reason for this modification comes from the specification of a terminating state in SIMP, which results in the output of the keyword *skip*. The definition of a command *skip* in our encoding would create a logic program that eventually returns the value skip, yet it is unable to differentiate between value and command, applying the command over the returned value again, and thus leading to an infinite recursive call of the command *skip*.

So far we have described the encoding of terms in LF, the operational semantics of primitive types, the family type *exp* as well as providing an overview of the memory representation. To complete the encoding of SIMP we must proceed by defining the operators on the heap memory and the structural operational semantics illustrated in section 4.1.2.

Prior to that, we must represent a signature that functions as typing checker from expressions to lower abstractions. Table 5.4 shows the representation of this model, where the signature *value* is instantiated with atomic expressions and thus canonical forms.

value : *exp* → *type*.
value/Int: value (int N).
value/Bool: value (prop B).
value/skip: value (skip).
value/loc: value (loc L).

Table 5.4: Signature of *value*

This signature becomes useful when describing evaluation semantics. The progress theorem, part of any type-safe language, states that a program is not stuck if it can be evaluated further or it is in canonical form. Therefore this signature tests whether the program logic is in canonical form or else it must continue its search for the following evaluation step.

5.2.3 Memory Access Functions

The encoding of SIMP continues by defining the semantics of the heap memory. Table 5.5 illustrates this signature, which is designed as an abstract list with two constructors, the expression *nil/exp*, representing a list with no further elements, and the expression *cons/exp* that forms a list with an element and a reference to another list. Hence our memory list is a recursively constructed succession of lists containing one element. An example: the SIMP heap memory containing expressions *true* and *one* is denoted in our encoded *heaplist* as *cons/exp* (*prop true*) (*cons/exp* (*int suc zero*) *nil/exp*), where *prop true* is an expression of type Boolean at index 0, *int suc zero* is an expression of type integer at index 1, and *nil/exp* is the end of the list, which cannot be indexed.

This example also highlights the order of addition of new elements in the list, where a new expression in the list is always added at the end of the list, represented by *nil/exp*, and not at the beginning. This is a design choice.

Access and modification operations in Table 5.5 work also by structural induction over its terms as previously explained when discussing the family type *plus* in section 5.2.1.

In the type declaration of $find\_exp$, a function to find an expression at a given location, the type terms *heaplist* and *nat* are expected as inputs, in order to locate an expression E, to be returned as output. The function term is recursively called on S:*heaplist* until the value N:*nat* (part of the expression constant *loc* N) is successfully unified with the base case, represented by *zero*, and thus delivering its content, the expression E. It is instrumental to understand that the zero of the base case is not index 0 in S but the result of recursively backtracking N times, starting from the very first element added to the list at index 0, and ending at the position index N. This means that new expressions added to the list do not modify any of the already existing indexes. For instance:
Having a heaplist S = *cons/exp skip* (*cons/exp prop true*(
cons/exp int zero nil/exp)) and a memory location *loc suc zero*, results in the following evaluation logic when applied to $find\_exp\ S\ L\ E$: expression *prop true* is returned as E after indexing first the element *skip* at location *suc zero* and completing the search arriving at position *zero*. A formal representation in Twelf is displayed in the Testing section on 6.

The *append_exp* signature takes an existing heaplist S and an expression E as input, returning both the location where it has been stored and a mod-

heaplist : *type*. %*name heaplist S*.

nil/exp: heaplist.
cons/exp: exp→ *heaplist* → *heaplist*.

find_exp: heaplist→ *nat* → *exp* → *type*.

find_exp/zero: find_exp (cons/exp E S) zero E.

find_exp/suc: find_exp (cons/exp E' S) (suc L) E
← *find_exp S L E*.

append_exp: heaplist→ *exp* → *heaplist* → *nat* → *type*.

append_exp/zero: append_exp nil/exp E (cons/exp E nil/exp) zero.

append_exp/suc: append_exp (cons/exp E' S) E (cons/exp E' S') (suc L)
← *append_exp S E S' L*.

replace_exp: heaplist→ *nat* → *exp* → *heaplist* → *type*.

replace_exp/zero: replace_exp (cons/exp E' S) zero E (cons/exp E S).

replace_exp/suc: replace_exp (cons/exp E' S) (suc L) E (cons/exp E' S')
← *replace_exp S L E S'*.

typeCheck: exp→ *exp* → *type*.

id/Int: typeCheck (int N) (int N').

id/prop: typeCheck (prop B) (prop B').

id/skipL: typeCheck (skip) (E).

id/skipR: typeCheck (E) (skip).

Table 5.5: Signature of heap memory: *heaplist*

ified copy of heaplist S, named S'. This is particular to our heaplist implementation design, where each newly add element has index N constructing a list of $(N + 1)$ elements. Hence, to calculate index N, *append_exp* recursively copies each element from S into S', counting the number of elements in order to calculate index N at the last position, and then adding the new element E. This method is time and memory consuming, however, it does not affect the goals of this project and consequently we do not consider relevant its improvement at this point.

The *replace_exp* signature is instantiated by three arguments of type S : *heaplist*, N : *nat*, and E : *exp* taken as input, an a meta-variable S' that constitutes the output result, produced by backtracked unification that systematically copies each element of S in S' until index N is reached, as this point it substitutes the stored expression E' with E while appending the remaining of S to S'. This signature works as the *modus operandi* of the encoded *assigns* command.

The *typeCheck* signature is also part of the *assigns* command. The heaplist is polymorphic and *replace_exp* does not distinguishes among expression types; this signature compares both expressions E and E' expecting both to be of the same canonical type or at least one of them of kind *skip*, here functioning as a neutral expression applied before a possible change of

type in a particular existing location, or to cancel a value. It is similar in use to the keyword $NULL$ in many imperative languages.

5.2.4 Structural Operational Semantics

Tables 5.6 and 5.7 represent a small example of the signature representing the reduction relation in SIMP. We have named this family type as *eval*. *Eval* is a representation of a computational step starting from an initial configuration $\langle S,\ E\rangle$ to a final one $\langle S',\ E'\rangle$, where $S,\ S'$ are meta-variables in our encoding instantiated by a heaplist, and $E,\ E'$ are also meta-variables instantiated by expressions,either compound, holding further reduction steps, or atomic, holding a canonical form. Consequently, the type *eval* takes the first two arguments as inputs, producing an output that depends on the inference rules applied to these arguments, instantiating the second pair of meta-variables. A pair $S,\ S'$ holds an identical heaplist when $S = S'$ after a reduction step which does not modify the memory, however, $E \neq E'$ at any time, as a reduction step is expected with every transition.

We have reduced considerably the number of elements representing the full signature of *eval* and kept only those that need further study. A description of the general behavior of *eval*'s derivations, as displayed in Table 5.6 and described below, is enough to understand the remaining derivations. For a complete description, we direct the reader to section A.1.

The reduction relation reduces a compound expression one computational step at a time. It recognises commands with the highest precedence and reduces its argument(s) to a canonical form in order for the command to be applied. This requires the relation to be recursively evaluated until an atomic expression is reached. Hence, To encode the reduction relation we must define a sequence of steps similar to that described in SIMP in section 4.1.2.

$eval : heaplist \rightarrow exp \rightarrow heaplist \rightarrow exp \rightarrow type.$	
$add/E1 : eval\ S\ (E1 + E2)\ S'\ (E1' + E2)$ $\quad \leftarrow eval\ S\ E1\ S'\ E1'.$	$\frac{\langle E_1,s\rangle \rightarrow \langle E_1',s'\rangle}{\langle E_1\ op\ E_2,s\rangle \rightarrow \langle E_1'\ op\ E_2,s'\rangle}$
$add/E2 : eval\ S\ (E1 + E2)\ S'\ (E1 + E2')$ $\quad \leftarrow value\ E1$ $\quad \leftarrow eval\ S\ E2\ S'\ E2'.$	$\frac{\langle E_2,s\rangle \rightarrow \langle E_2',s'\rangle}{\langle E_1\ op\ E_2,s\rangle \rightarrow \langle E_1\ op\ E_2',s'\rangle}$
$add/comp : eval\ S\ (int\ N1 + int\ N2)\ S'\ (int\ N3)$ $\quad \leftarrow plus\ N1\ N2\ N3.$	$\frac{}{\langle n_1\ op\ n_2,s\rangle \rightarrow \langle n,s\rangle,\ if\ n=(n_1\ op\ n_2)}$

Table 5.6: General encoding of small-step semantics

For the derivation of the command + in Table 5.6, we have provided the original definition in SIMP and the encoding in Twelf as a way of illustrating the general behavior of each relation.

We begin by defining a relation $add/E1$ where the first argument $E1$ of the command + is reduced by calling again on the *eval* relation, this time only with $E1$ and S : *heaplist* as input, returning $E1'$ and S'. Then, it follows by declaring a definition of $add/E2$ where the second argument $E2$ is reduced in the same fashion as $E1$, provided $E1$ is already in canonical form, denoted by the term *value* $E1$. Therefore, the unification algorithm is only successfully applied to this case when a proof of $valueE1$ can be constructed. Finally, when both arguments are reduced to their atomic expressions, illustrated in *add/comp* as *int* $E1$ and *int* $E2$, we apply the correspondent primitive operation to their canonical values, this generates the term *plus* $N1$ $N2$ $N3$, where $N1$ and $N2$ are given by the program specification, returning the result value $N3$ that forms the output expression $intN3$. An alternative version has a command with one argument only (as command ! in Table 5.7). In this case it simply must reduce the expression argument to its canonical value, if necessary, followed by the computational step required.

The command *assigns*, in Table 5.7, updates a location in the heaplist with a new expression. For such result, it checks whether the expression to be updated is in canonical form, if succeeds, it continues unifying with the type below, finding the expression already at location L and checking it is the same type as the expression to be updated with. Finally, it replaces the expression E' with the new expression E, producing an output of value *skip*. This is essential for the sequencing command to reduce terms and parse the program, and the main reason behind the application of this command right after the command ref. Failure to do so results in an output of *loc* L, halting the program.

The sequencing derivation of command *";"* shows how the program is reduced towards a terminating configuration. It basically expects two *skip* expressions at both sides to reduce them to one *skip* expression, and so on and so forth until the program consists of a single *skip* term. We say that it *expects* two *skip* expressions because we do not force the atomic expressions to be *skip* after checking it is in canonical form. If we did so, there is always a chance of reducing an expression to a canonical form that it is not *skip* and the parser would continue anyway. However this in an incorrect derivation not allowed by the original definition of SIMP. Hence, by expecting a *skip*, yet not forcing it, we have built a tester for erroneous terminating transitions. Exceptionally, the very last command applied after the final *";"* is able to produce and print a result distinct from *skip*. We have decided not to modified this action as it helps us to test different operators and their results in the testing section at 6.

The conditional command if does not modify the memory when applied; it only decides between two possible paths depending of a Boolean result. Therefore we define the initial an final memory state with the same meta-variable S. Having described the command with the parameter prefix, it

is necessary to encapsulate both the command and the Boolean expression between brackets in order for Twelf to implement them successfully. This is due to Twelf not recognising a prefix with more than one argument.

We have not yet provided a definition for the semantics of if. It is a simple signature described below.

$$if\_then\_else : \ bool \rightarrow exp \rightarrow exp \rightarrow exp \rightarrow type.$$

$$if\_then\_else/tt : \ if\_then\_else \ true \ E1 \ E2 \ E1.$$

$$if\_then\_else/ff : \ if\_then\_else \ false \ E1 \ E2 \ E2.$$

Notice how the Boolean operand is reduced from an expression to a base type before reaching this signature, however, we are not interested in reducing any of the two expressions prior to knowing which path the program must follows. The reason is that if both expressions are evaluated and one produces an error or an infinite loop, regardless of whichever path we follow, the program will not reach a terminating state.

$bang/E : eval \ S \ (!\ E1) \ S' \ (!\ E1')$
$\quad \leftarrow eval \ S \ E1 \ S' \ E1'.$

$bang/comp : eval \ S \ (!\ loc \ L) \ S \ E$
$\quad \leftarrow find\_exp \ S \ L \ E.$

$reference/E : eval \ S \ (ref \ E) \ S' \ (ref \ E')$
$\quad \leftarrow eval \ S \ E \ S' \ E'.$

$reference/comp : eval \ S \ (ref \ E) \ S' \ (loc \ L)$
$\quad \leftarrow value \ E$
$\quad \leftarrow append\_exp \ S \ E \ S' \ L.$

$update/E1 : eval \ S \ (E1 \ assigns \ E2) \ S' \ (E1' \ assigns \ E2)$
$\quad \leftarrow eval \ S \ E1 \ S' \ E1'.$

$update/E2 : eval \ S \ (E1 \ assigns \ E2) \ S' \ (E1 \ assigns \ E2')$
$\quad \leftarrow value \ E1$
$\quad \leftarrow eval \ S \ E2 \ S' \ E2'.$

$update/comp : eval \ S \ (loc \ L \ assigns \ E) \ S' \ (skip)$
$\quad \leftarrow value \ E$
$\quad \leftarrow find\_exp \ S \ L \ E'$
$\quad \leftarrow typeCheck \ E' \ E$
$\quad \leftarrow replace\_exp \ S \ L \ E \ S'.$

$if/B : eval \ S \ ((if \ B) \ E1 \ E2) \ S \ ((if \ B') \ E1 \ E2)$
$\quad \leftarrow eval \ S \ B \ S \ B'.$

$if/comp : eval \ S \ ((if \ prop \ B) \ E1 \ E2) \ S \ (E3)$
$\quad \leftarrow if\_then\_else \ B \ E1 \ E2$

$while/ : eval \ S \ ((while \ B) \ E) \ S \ (E1)$
$\quad \leftarrow eval \ S \ ((if \ B) \ (E \ ; \ ((while \ B) \ E)) \ skip) \ S \ E1.$

$sequencing/E1 : \ eval \ S \ (E1 \ ; \ E2) \ S' \ (E1' \ ; \ E2)$
$\quad \leftarrow \ eval \ S \ E1 \ S' \ E1'.$

$sequencing/E2 : \ eval \ S \ (E1 \ ; \ E2) \ S' \ (E1 \ ; \ E2')$
$\quad \leftarrow \ value \ E1$
$\quad \leftarrow \ eval \ S \ E2 \ S' \ E2'.$

$sequencing/comp : \ eval \ S(skip \ ; E2) \ S' \ (E2)$
$\quad \leftarrow \ value \ E2.$

Table 5.7: Encoding of small-step semantics: commands

The command *while* is described as syntactic sugar for the command if in SIMP, and thus we reproduce here this similarity. It also needs brackets around the Boolean argument, as we have declared it with a prefix parameter.

When correctly applied, the *eval* signature reduces a configuration one computational step. From an initial configuration $\langle E1,\ S\rangle$ we aim to reduce to a configuration $\langle E1',\ S\rangle$, where $E1'$ is of canonical form, so that a computation can be done on the value, producing a result. However, if the initial configuration needs more than one reduction step to reach a final configuration, our language is unable of producing, at this moment, more than one consecutive steps. The solution is to implement the evaluation relation described in section 4.1.2 as part of the SIMP grammar.

Looking at the grammar, we observe that each of the relations contains a sequence of reduction steps that leads to a terminating state. All of these commands have already been previously defined, thereby we must built a judgement capable of recursively calling each of the necessary reduction steps to reach a final configuration. Table 5.8 shows this encoding.

$$run : \ heaplist \rightarrow exp \rightarrow heaplist \rightarrow exp \rightarrow type.$$

$$\begin{aligned} run/suc : \ & run\ S\ E\ S1\ E1 \\ & \leftarrow eval\ S\ E\ S2\ E2 \\ & \leftarrow run\ S1\ E1\ S2\ E2. \end{aligned}$$

$$run/zero : \ run\ S\ E\ S\ E.$$

Table 5.8: Encoding of Evaluation relation

The difference compared to other relations described in this design is that the base case is placed *after* the recursive call. *Run/suc* only terminates unifying when it encounters an expression term in *eval* that it is not defined as one of its rules, therefore, it terminates when the output is incorrect or a canonical value, unifying then with the base case *run/zero* and halting. Providing a definition of skip as a command would have then led to an infinite set of evaluation steps, due to the recursive nature of *run*. Hence our alternative design where skip has type expression.

In this section we have provided an encoding of the grammar of SIMP. We have also described the use of the different keywords encountered while building the specification of the language and how they affect our encoding. The design of SIMP has been by directly implemented in Twelf,remarking on the design choices we have made.

We now conclude the main body of the project by testing the encoded language with a few examples, followed by the implementation and testing

of an algorithm in the mechanised logic of the language.

6 Testing

We use Twelf built-in software to prove properties about the code we just defined by using in its specification language. To this purpose, there are different keywords achieving different results. We are interested in the keyword `%query`. `%query` provides as many different solutions as requested by the given parameters(as long as there are as many solutions to give). In a solution, for each of the meta-variables occurring in a given program, `%query` provides a single definition. It can also produce a witness proof of the solution if requested.

Before illustrating this far with an example, it is necessary to understand how the Twelf server works. Firstly we must configure the server with a file ending in *.cfg*. This configuration file contains each of the .elf files we need for our encoding to run. All files must be written in order for Twelf to implement them, thus if an specification of *nat* is in file A and an specification of *exp* is in file B, knowing that *exp* builds on top of *nat*, we must write file A's name before file B's in *.cfg*.

After linking the *.cfg* file in Twelf, we look for the server file, known by default as **Twelf-server**. When Emacs displays `OK`, we must *check configuration*, this implements our logical system in Twelf, as long as it is correct, and Twelf illustrates this by displaying the code contained in each *.elf* file as well as inferring the dependent types of each relation and explicitly displaying them before each term. If the whole procedure terminates successfully, Twelf closes the *.elf* files and prints `OK` (see section A.2 for a printed version). Then we are ready to query the server.

Below we have defined some syntactic sugar for integers and Boolean, as well as naming some of the memory locations we use in our testing script, these are the variables of the language. By means of this we aim to reduce the writing process and make the derivations easier to follow.

```
 0 = int zero.
 1 = int (suc zero).
 2 = int (suc suc zero).
 3 = int (suc suc suc zero).
 4 = int (suc suc suc suc zero).
 5 = int (suc suc suc suc suc zero).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tt = prop true.
ff = prop false.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
a = loc zero.
b = loc suc zero.
c = loc suc suc zero.
n = loc suc suc suc zero.
```

We start by testing/querying the arithmetic operators, we use big-step semantics as our program encoder, describing as input the initial memory and the program logic itself. This first example adds two numbers, 3 and 4, expecting a result of 7. We give two parameters to `%query`, the first asks to check whether there are at least as many solutions as the given value, and the second one displays as many solutions as the argument value. We expect one answer as we have designed a deterministic language, so we request 1, however, there are more than one solution, as each evaluation step is taken as a possible intermediate answer. We use this at our advantage as shown in later examples:

```
%query 1 1 run nil/exp (3 + 4) S E.
```

We do not need the memory this far, thus we construct an empty memory. A meta-variable instead of `nil/exp` would also work, but `nil/exp` is a more *tangible* description of the memory than just a variable. The solution is reproduced below:

```
readDecl
%query 1 1 run nil/exp (3 + 4) S E.%.
%query 1 1

run nil/exp (3 + 4) S E.
---------- Solution 1 ----------
E = int (suc suc suc suc suc suc suc zero);
S = nil/exp.
-------------------------------------------

%% OK %%
```

Notice that the result of E is valid as testing, but invalid as output of SIMP. A valid terminated state in SIMP is expressed by *skip*. Next, we multiply 2 times 5, expecting a result of 10.

```
readDecl
%query 1 1 run nil/exp (2 * 5) S E.%.
%query 1 1

run nil/exp (2 * 5) S E.
---------- Solution 1 ----------
E = int (suc suc suc suc suc suc suc suc suc suc zero);
S = nil/exp.
-------------------------------------------

%% OK %%
```

To end simple arithmetic operations, we divide 5 by 3, resulting in 1.

```
readDecl
%query 1 1 run nil/exp (5 / 3) S E.%.
%query 1 1

run nil/exp (5 / 3) S E.
```

```
---------- Solution 1 ----------
E = int (suc zero);
S = nil/exp.
-------------------------------------------

%% OK %%
```

Let's try something more complex, what is the result of `(2 + 5 - 1 * 3)`? As we have previously declared that the precedence of a multiplication is higher than that of a rest and both operators are left associative, this operation is the same as `(2 + 5)-(1 * 3) = 4`. To demonstrate it, we ask the query to give us as many solutions as possible by writing `%query 1 *`. This particular expression proves to be a valuable command to construct proofs of our programs. By means of asking the server to produce an infinite number of solutions (the parameter `*`) providing there is at least 1 solution to the program logic (the parameter `1`), we are building a reduction transition based on the given grammar and the fact that our language is deterministic.

```
readDecl
%query 1 * run nil/exp (2 + 5 - 1 * 3) S E.%.
%query 1 *

run nil/exp (2 + 5 - 1 * 3) S E.
---------- Solution 1 ----------
E = int (suc suc suc suc zero);
S = nil/exp.
---------- Solution 2 ----------
E = int (suc suc suc suc suc suc suc zero) - int (suc suc suc zero);
S = nil/exp.
---------- Solution 3 ----------
E = int (suc suc suc suc suc suc suc zero) - 1 * 3;
S = nil/exp.
---------- Solution 4 ----------
E = 2 + 5 - 1 * 3;
S = nil/exp.
stdIn:1.2-1.44 Error:
Query error -- wrong number of solutions: expected 1 in * tries, but found 4
%% ABORT %%
```

Although it outputs `ABORT` at the end of the file, it produces the expected result as well as a proof step by step of how it arrives to such value. `ABORT` is here more a warning, we ask `%query` to print infinite solutions with the value `*`, as we did not know how many steps it would take to reach a terminating state, but, obviously, it has a finite number of solutions, therefore the keyword `ABORT`.

Then, what answer does it produce if we add some inner parentheses? For instance, what is the solution to `((2 + 5 - 1)* 3)` ?In this case it should be 18 and not 4 as before. Let's check.

```
readDecl
%query 1 * run nil/exp ((2 + 5 - 1) * 3) S E.%.
%query 1 *

run nil/exp ((2 + 5 - 1) * 3) S E.
```

```
---------- Solution 1 ----------
E =
   int
      (suc suc suc suc suc suc suc suc suc suc suc suc suc suc suc suc suc suc
          zero);
S = nil/exp.
---------- Solution 2 ----------
E = int (suc suc suc suc suc suc zero) * 3;
S = nil/exp.
---------- Solution 3 ----------
E = (int (suc suc suc suc suc suc suc zero) - 1) * 3;
S = nil/exp.
---------- Solution 4 ----------
E = (2 + 5 - 1) * 3;
S = nil/exp.
stdIn:1.2-1.46 Error:
Query error -- wrong number of solutions: expected 1 in * tries, but found 4
%% ABORT %%
```

Indeed the answer we were looking for. To conclude we test what happens if we include a Boolean type in an arithmetic operation. We divide 4 by 2, add 3 and multiply by `tt`.

```
readDecl
%query 1 * run nil/exp ((4 / 2 + 3) * tt) S E.%.
%query 1 *

run nil/exp ((4 / 2 + 3) * tt) S E.
---------- Solution 1 ----------
E = int (suc suc suc suc suc zero) * tt;
S = nil/exp.
---------- Solution 2 ----------
E = (int (suc suc zero) + 3) * tt;
S = nil/exp.
readDecl
%query 1 1 run nil/exp ((4 / 2 + 3) * tt) S E.%.
readDecl
%query * * run nil/exp ((4 / 2 + 3) * tt) S E.%.
readDecl
%query 1 * run nil/exp ((4 / 2 + 3) * tt) S E.%.
```

We notice that the first time it gets to perform the arithmetic operations until if find `tt`, and then stops. If we query the server again with different parameters, it halts and becomes unable to give any answer.

We now test the Boolean operators. As we have seen the inner workings of Twelf, we just reproduce a list with all the testing code that has been done.

```
%query 1 1

run nil/exp (tt ^ tt) S E.
---------- Solution 1 ----------
E = prop true;
S = nil/exp.
-----------------------------------------
```

```
%query 1 1

run nil/exp (tt ^ tt ^ ff) S E.
---------- Solution 1 ----------
E = prop false;
S = nil/exp.
-------------------------------------------

%query 1 1

run nil/exp (2 < 3) S E.
---------- Solution 1 ----------
E = prop true;
S = nil/exp.
-------------------------------------------

%query 1 1

run nil/exp (2 > 3) S E.
---------- Solution 1 ----------
E = prop false;
S = nil/exp.
-------------------------------------------

%query 1 1

run nil/exp (2 == 3) S E.
---------- Solution 1 ----------
E = prop false;
S = nil/exp.
-------------------------------------------

%query 1 1

run nil/exp (0 == 0) S E.
---------- Solution 1 ----------
E = prop true;
S = nil/exp.
-------------------------------------------

%query 1 1

run nil/exp (2 > 3) S E.
---------- Solution 1 ----------
E = prop false;
S = nil/exp.
-------------------------------------------

%query 1 1

run nil/exp (not tt) S E.
---------- Solution 1 ----------
E = prop false;
```

```
S = nil/exp.
-----------------------------------------

%query 1 1

run nil/exp (not ff) S E.
---------- Solution 1 ----------
E = prop true;
S = nil/exp.
-----------------------------------------

readDecl
%query 1 1 run nil/exp (tt ^ 2) S E.%.
%query 1 1

run nil/exp (tt ^ 2) S E.
---------- Solution 1 ----------
E = tt ^ 2;
S = nil/exp.
-----------------------------------------

%% OK %%
readDecl
%query 1 1 run nil/exp (2 < ff) S E.%.
%query 1 1

run nil/exp (2 < ff) S E.
---------- Solution 1 ----------
E = 2 < ff;
S = nil/exp.
-----------------------------------------

%% OK %%
readDecl
%query 1 1 run nil/exp (2 > tt) S E.%.
%query 1 1

run nil/exp (2 > tt) S E.
---------- Solution 1 ----------
E = 2 > tt;
S = nil/exp.
-----------------------------------------

%% OK %%
readDecl
%query 1 1 run nil/exp (2 == ff) S E.%.
%query 1 1

run nil/exp (2 == ff) S E.
---------- Solution 1 ----------
E = 2 == ff;
S = nil/exp.
-----------------------------------------
```

```
%% OK %%
readDecl
%query 1 1 run nil/exp (not 2) S E.%.
%query 1 1

run nil/exp (not 2) S E.
---------- Solution 1 ----------
E = not 2;
S = nil/exp.
-----------------------------------------

%% OK %%
```

We follow our testing by applying the distinct commands that access the memory.
The first example shows the reduction steps of memory modification. We store two values in the heap at index 0 and 1, to follow by updating index 0 with the sum of the value at index 0 and at index 1. The initial values are 3 and 5, stored at index 0 and 1 respectively, and the final values are, at index 0, value 8 and, at index 1, value 5.

```
readDecl
%query 1 * run nil/exp (ref skip assigns 3 ; ref skip assigns 5 ;
 index0 assigns ! index0 + ! index1) S E.%.
%query 1 *

run nil/exp
   (ref skip assigns 3 ; ref skip assigns 5 ; index0 assigns ! index0 +
! index1)
   S E.
---------- Solution 1 ----------
E = skip;
S = cons/exp (int (suc suc suc suc suc suc suc suc zero)) (cons/exp 5 nil/exp).
---------- Solution 2 ----------
E = skip ; skip;
S = cons/exp (int (suc suc suc suc suc suc suc suc zero)) (cons/exp 5 nil/exp).
---------- Solution 3 ----------
E = skip ; index0 assigns int (suc suc suc suc suc suc suc suc zero);
S = cons/exp 3 (cons/exp 5 nil/exp).
---------- Solution 4 ----------
E = skip ; index0 assigns 3 + 5;
S = cons/exp 3 (cons/exp 5 nil/exp).
---------- Solution 5 ----------
E = skip ; index0 assigns 3 + ! index1;
S = cons/exp 3 (cons/exp 5 nil/exp).
---------- Solution 6 ----------
E = skip ; index0 assigns ! index0 + ! index1;
S = cons/exp 3 (cons/exp 5 nil/exp).
---------- Solution 7 ----------
E = skip ; skip ; index0 assigns ! index0 + ! index1;
S = cons/exp 3 (cons/exp 5 nil/exp).
---------- Solution 8 ----------
E = skip ; loc (suc zero) assigns 5 ; index0 assigns ! index0 + ! index1;
S = cons/exp 3 (cons/exp skip nil/exp).
---------- Solution 9 ----------
```

```
E = skip ; ref skip assigns 5 ; index0 assigns ! index0 + ! index1;
S = cons/exp 3 nil/exp.
---------- Solution 10 ----------
E = loc zero assigns 3 ; ref skip assigns 5 ; index0 assigns ! index0 +
! index1;
S = cons/exp skip nil/exp.
---------- Solution 11 ----------
E = ref skip assigns 3 ; ref skip assigns 5 ; index0 assigns ! index0 +
! index1;
S = nil/exp.
stdIn:1.2-1.107 Error:
Query error-- wrong number of solutions: expected 1 in * tries, but found 11
%% ABORT %%
```

Notice how it starts by changing back our naming conventions into their original values. Also, the lack of parentheses due to the specified order of precedence (see section 5.2.2), where the sequencing command has the lowest, thus it only operates when both arguments on each side of the command are reduced to the canonical expression *skip*. This is the main reason on declaring a local variable with a value and assigning an extra value at the same time, as seen in the expression: $ref\ skip\ assigns\ 3$. If we only declare the variable with an initial value, by applying $ref\ skip$, this value is stored, and its location in the heaplist is returned, however, it would stop the reduction process of the program, as the sequencing command only evaluates skip expressions.

6.1 Encoding of the Fibonacci Algorithm

Having favourably test most of the encoding of SIMP, we now aim to create a program that applies each of the terms of our encoding an successfully produces the expected answer. For this matter we have decided to encode the *Fibonacci numbers* algorithm as it presents us with a well known logic program that we can prove in writing beforehand.

The Fibonacci numbers is, by definition, a sequence with the first two numbers in the Fibonacci series defined as 0 and 1, and each subsequent number is the sum of the previous two.

$$Fib(n) = \begin{cases} Fib(0) & = & 0 \\ Fib(1) & = & 1 \\ Fib(n \geq 2) & = & Fib(n-1) + Fib(n-2) \end{cases}$$

Some Fibonacci numbers are: $Fib(2) = 1,\ Fib(3) = 2,\ Fib(4) = 3$.

We have designed an algorithm using the grammar of SIMP that iteratively produces $Fib(n)$, this is:

```
Fib(n: integer) returns integer a.
begin
  a := 0;
  b := 1;
```

```
  while (n > 0)
   {
    c := a + b;
    a := b;
    b := c;
    n := n - 1;
   }
end.
```

Our aim is to encode the SIMP algorithm in our specified language and produce valid results as well as a proof of the validity of the results. As the proof witness entails a very large amount of data, we have decided to produce in this section an example without formal proof or the distinct 157 solutions that constitute each of the computational steps.Instead, we directly deliver the final answer to our query. For an additional example with a valid formal proof, we direct the reader to the appendix in section A.3.

Hence, by using the $Fib(n)$ algorithm, what is the third Fibonacci number, $Fib(n = 3)$ in SIMP?:

Given a configuration `<P,s>` where `P` is the program `a := 0; b:= 1; n:= 3; while (n > 0)(c := a + b; a := b; b:= c; n := n -1)`, there is a unique evaluation sequence of transitions with maximal length that reaches a terminal configuration. This configuration is:
$\langle P, s\rangle$
$\mapsto \langle a := 0; b := 1;\ n := 3;\ while\ (!n > 0)\ (c := !a + !b;\ a := !b;\ b := !c;\ n := !n - 1), s\rangle$
$\mapsto^* \langle skip, s[a \mapsto 2;\ b \mapsto 3;\ n \mapsto 0;\ c \mapsto 3]\rangle$.

And the reduction steps are schematically described here as:

$while(3 > 0)$	$while(2 > 0)$	$while(1 > 0)$	$while(0 > 0)$
$true$	$true$	$true$	$false$
$c := 1$	$c := 2$	$c := 3$	$return\ a$
$a := 1$	$a := 1$	$a := 2$	
$b := 1$	$b := 2$	$b := 3$	
$n := 2$	$n := 1$	$n := 0$	

where `a = 2` and thus the expected result.

Having shown that our algorithm successfully produces the expected answer, we must encode it in our specified grammar and prove that correctly outputs a valid answer. Below we reproduce the output of Twelf, containing the encoded algorithm plus the final solution. Notice that the four assignments at the beginning of the program correspond to variables `a, b, c` (with a null value), and `n` respectively.

```
readDecl
%query 1 1

run nil/exp
   (ref skip assigns 0 ; ref skip assigns 1 ; ref skip assigns skip
       ; ref skip assigns 3
       ; (while ! n > 0)
            (c assigns ! a + ! b ; a assigns ! b ; b assigns ! c
                ; n assigns ! n - 1)) S E.
---------- Solution 1 ----------
```

```
E = skip;
S =
   cons/exp (int (suc suc zero))
      (cons/exp (int (suc suc suc zero))
          (cons/exp (int (suc suc suc zero)) (cons/exp (int zero) nil/exp))).
------------------------------------------

%% OK %%
```

The program outputs the expression `skip` as part of the terminating configuration and it contains a memory `S` with index 0 = integer 2 (variable a), index 1 = integer 3 (variable b), index 2 = integer 3 (variable c), index 3 = integer 0 (variable n).

Therefore, the language encoded in Twelf from an specification of SIMP is correct with respect to the results it produces. By using Twelf we are able to guarantee that any of the results originating from this code are valid. However, we are not able to guarantee the isomorphism of the implementation and SIMP until formally proved.

7 Evaluation

By the end of this project we expected to build an isomorphic representation of the language SIMP in LF, by means of using Twelf, and prove properties about the language by building a program capable of producing a valid solution based in the grammar of SIMP. The encoding has provided a solution acceptable in terms of expected result values. The generated proof in appendix A.3 shows each of the reduction steps computed in order to achieve the answer to the Fibonacci algorithm as well as a witness proof term consisting on each of the recursive calls made during the evaluation of the program. This should prove sufficient to consider our encoding as adequate to SIMP, however, the adequacy proof is not strong enough as each of the testing examples should have been directly compared with the transition system of SIMP, instead, it was compared against mathematical equations.

All the testing examples have terminated in a valid state, and we expect the transition sequence to be equivalent to the transition sequence of SIMP. For this matter we should have provided a set of configurations to be compared against the set of configurations printed by Twelf, and not relay only on final results.

This became evident at the end of the process cycle, when testing the algorithm provided. Although a valid answer was reached when querying the language, the proof accompanying it illustrated that there were a series of different paths tested before arriving to the expected answer. These paths tested the different alternative solutions to the Boolean operation `(!n > 0)` within the conditional; `if`, branching from both the `true` and

`false` conditions. Later we discovered that the definition of the Boolean operators were not strong enough and needed extra constraints, particularly the operators *greater than* and *equals to*. This branching was due to the inner algorithm of Twelf, which inductively would try to unify with any possible derivation available, regardless of whether it had already found a previous valid unification, yet always taking for a valid unification the base case closest to the type definition of the term. This fault affects to the signature of `and`, providing three constants `and true true` when the parameters are both true.

Hence, to ensure that our implementation of SIMP is identical to the original specification, and not just that these particular testing examples run appropriately, we produce a proof of adequacy. One of the methods would be to compare our implementation relative to a distinct implementation made of SIMP in Twelf. This would be done by comparing proofs created by both encodes. However, being SIMP a toy language used for the study of imperative languages make the chances of finding an alternative model low. In this case, as SIMP is also a logical system, we would define by hand an alternative model and proof both their semantics are equivalent. This is the method used by Harper et al. in [9]

Other issues were, the declaration of commands if and while also created a difficult environment, particularly because of their definition as prefix commands. Twelf did not allow a normal evaluation implementation of both commands and much time was spent on trying to find out an alternative. Finally the placement of brackets around the command and the first argument proved successful, yet it complicated the sequencing scenario, where now we could encounter and expression `skip ; ( skip ;...)` not producing computational step between these skip expressions until another skip was evaluated on the right hand side of the bracket.

On a different scenario we would omit the keywords `%prefix, %infix` as it created more difficulty than benefits, leaving each command as a prefix term. This would have saved time spent on finding the right precedence value for each of the terms of the language, which it proved at times awkward and hard to guess. Giving a higher precedence to commands, and particularly the sequencing command, resulted in the program halting at every evaluation when consisting of many different commands and expressions. This was due to the algorithm being unable to distinguish where to start from, and which arguments belong to each of the commands. When changing the precedence conversely, the program worked without failures, yet by reducing each of the arithmetic operators and referencing commands first, it made nearly unnecessary the reduction relation of any other command with two arguments apart from the sequencing command. This was first experienced when looking at the output generated by the final example, where evaluation computations and sequencing reductions form the majority of the proof.

These are, however, minor problems that could be easily solved in a fu-

ture revision of the project. We find the body of the project satisfactory and meeting the required specifications to a high level. We were able to produce a logical language capable of imitating the behavior of an imperative language, including a description of its memory, which it was one of the main challenges to achieve in this project. Some of the design choices were not optimal, as for instance the storing of a value in the heaplist, which it involved and exhaustive search each time a location was requested. Storing each new value at index 0 and creating an additional relation to count the number of constants in the heaplist would have been a faster and more effective choice. Having said that, our project was not based on using effectively resources but on satisfactorily implementing a type-safe and memory-safe language, and that, we did achieve.

One of the main difficulties of this project was the fact that there was not enough information readily available, and the Twelf user guide was outdated and written with little attention to detail, meaning that it was easier to follow user's examples updated in the Twelf site than the guide itself. One of the works that helped immensely and from which we developed our own code was the case study by Robert J. Simmons, named *mutable state. Encoding a simple imperative language with state.*[14] Our language enhances his existing one by providing a fully imperative language, while his provides commands and functionality specific of a functional language as well as an imperative one. Also we implement the a full version of the language, deriving each of the syntactic structures that compose an imperative language, while his encoding proves to be quite basic as he is more focus on describing proofs and properties of the language. We describe our language as fully operational, as it has been demonstrated by the examples on the Testing section.

The plan designed before the beginning of the project was followed to a certain extent, much time than necessary was consumed on trying to understand the logic behind the LF research papers, when a more practical approach, consisting on an earlier contact with the proof assistant, would have been more productive.

To conclude, we believe that the project here described meets the requirements previously specified, building an encode in LF of the toy language SIMP, capable of imitating its behavior while guaranteeing type-safety as well as memory-safety. This encode is then used as a parser within the program Twelf, where queries are correctly answered and proofs of the results are also delivered. However, further formal testing of SIMP should have been provided to compare step by step the semantics of both languages.

[14] http://twelf.org/wiki/Mutable_state

8 Conclusions

This project was intended as an example on how to build certified software, particularly the mechanisation of a meta-logic that allows the programmer to described any of the aspects of the program. By formalising the implementation we construct a means to prove the validity of the code and its properties by a machine, increasing its reliability and producing programs correct-by-construction, considerably reducing type safety issues.

In order to produce such implementation we have surveyed the various different logical frameworks available, deciding to encode the language SIMP in LF, as it provided sufficient expressiveness to described a deductive system as the one we were about to implement. The proof assistant Twelf was a logical choice as it is based in LF, therefore we could directly implement our LF design in the proof assistant as both LF and Twelf shared the same syntax. Other choices could have also done the work, however, Twelf provided us with a search and induction algorithm that was very similar to the transition system of SIMP, recursively reducing each computational term until reaching a base case.

With these tools we have created an encoding of SIMP in Twelf that shares functionality and properties, particularly type safety. This proves an achievement when dealing with unsafe languages, as we are able to describe the behavior and properties of the language and mathematically proving their validity. This proof can be used further to prove other properties, or if extending the language, it can once again test the validity of each of the terms that constitute the language.

By describing the design and implementation together in Twelf, we have also created an user guide for future researchers in the field. This is very valuable as the amount of information is very restricted, and the proof assistant user guide is outdated. Moreover, most of the examples in Twelf's website are related to functional languages and lambda-calculus, this work is one of the few that fully takes on the implementation of an imperative language and its side effects, delivering a parser capable of constructing programs and producing answers, both within the operational semantics of Twelf. The various examples implemented in the language showed that the encoding is able to produce a terminating state containing the expected answer in a logically-built heap memory. This encoding entails that the results would remain the same regardless of the operative software used.

Some commands have been left outside, for instance the command for, yet any of the commands left outside could be implemented by means of those already part of the encoding and therefore we find them unnecessary to include.

Future work should be directed, first, in building a formal proof of adequacy as explained in section **??**, and subsequently towards building theorem-like terms to prove properties of the language, such as associative,

effectiveness lemma, progress and preservation. For this purpose a level of types should also be declared, as well as a type list that mirrors the expressions in the heaplist. Our description of the type of expressions is limited and abstract, a family type of types in SIMP would be useful in order to provide formal proofs covering every layer of the language. It could follow by implementing a formal model of the hardware, in order to prove the correctness of machine instructions and thus finally creating the initially specified *Certified framework*.

References

[1] Tony Hoare, Jay Misra (2005). *Verified software: theories, tools, experiments. Vision of a Grand Challenge project*. Microsoft Research Ltd. and the University of Texas at Austin.

[2] Anthony Hall, Roderick Chapman (2002). *Correctness by Construction: Developing a commercial secure system*. pp18-25. IEEE Software publications, January/February.

[3] Robert Harper (2011). *Practical foundations for programming languages (version1.18)*. Carnegie Mellon University. Unpublished, Available at **URL**:*http://www.cs.cmu.edu/ rwh/plbook/book.pdf*.

[4] Xinyu Feng (2007). *An open framework for Certified System Software*. Ph.D dissertation, Yale University.

[5] Zhong Shao (2010). *Certified Software*. In Communications of the ACM, 53(12), pages 56-66. Yale University.

[6] Robert Harper, Daniel R. Licata (2007). *Mechanizing metatheory in a logical framework*. Journal of functional programming, v.17, n.4-5, p.613-673, Carnegie Mellon University

[7] David Basin, Manuel Clavel, and José Meseguer (2004). *Reflective metalogical frameworks*. Transactions on computational logic (TOCL), volume 5, Issue 3.

[8] Carsten Schürmann, Frank Pfenning (1998). *Automated theorem proving in a simple meta-logic for LF* In Claude Kirchner and Hélène Kirchner, editors, Proceedings of the 15th International Conference on Automated Deduction (CADE-15), pages 286300, Lindau, Germany. Springer-Verlag LNCS 1421.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin (1993). *A framework for defining logics*. Journal of the Association for Computing Machinery, 40(1):143184.

[10] Haskell B Curry, Robert Feys (1958). *Combinatory logic*, volume 1. North Holland, second edition, 1968.

[11] William A. Howard (1980, Reprint of 1969 article). *The formulas-as-types notion of construction*. In J.P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory logic, λ-calculus, and Formalism*, pages 479-490. Academic Press, New York.

[12] Benjamin C. Pierce (2002). *Types and programming languages* The MIT Press. Cambridge, Massachussetts and London, England. pp. 108-109.

[13] Dana S. Scott (1993). *A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science*, 121:411-440. Annotated version of the 1969 manuscript.

[14] Nicolaas G.de Bruijn (1994). *Reflections on Automath*. In Selected papers on Automata, Nederpelt, Geuvers, and de Vrijer, Eds. North-Holland, pp. 201-228.

[15] Nicolaas G. de Bruijn (1968). *AUTOMATH, a language for mathematics*. Eindhoven University of Technology. T.H. Report 66-WSK-05.

[16] Frank Pfenning (1996). *The practice of Logical Frameworks*. In Hélène Kirchner, editor, Proceedings of the Colloquium on Trees in Algebra and Programming, pp.119-134, Linkping, Sweden. Springer-Verlag LNCS 1059. Invited talk.

[17] Arnon Avron, Furio Honsell, Ian A. Mason and Robert Pollack (1992). *Using Typed λ-calculus to Implement Formal Systems on a Machine*. Journal of Automated Reasoning, Volume 9, pp. 309–354.

[18] Fairouz Kamareddine, Twan Laan and Rob Nederpelt (2003). *De Bruijn's AUTOMATH and Pure Type Systems*. In *applied logic series 28. Thirty-five years of automating mathematics*, pp. 71–124

[19] C. A. R. Hoare (1969). *An axiomatic basis for computer programming*, Communications of the ACM, v.12, n.10, pp.576-580.

[20] Michael J. C. Gordon (1989). *Mechanizing programming logics in Higher Order Logic*. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*, pp. 387–439, New York, USA. Springer-Verlag.

[21] G. Roşu, C. Ellison, and W. Schulte (2010). *Matching logic: An alternative to Hoare/Floyd logic*. In 13th International Conference on Algebraic Methodology and Software Technology (AMAST10).

[22] Frank Pfenning (1998). *Partial polymorphic type inference and higher-order unification*. In proceedings of the 1988 ACM conference on lisp and functional programming,pp. 153–163, Snowbird, Utah, USA.

[23] Yves Bertot, Pierre Castran (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Series: Texts in theoretical Computer Science. An EATCS series, XXV, pp. 47–61, Springer-Verlag.

[24] Frank Pfenning (1991). *Logic programming in the LF logical framework*. Cambridge University press, New York, USA.

[25] Benjamin C. Pierce (2005). *Advanced Topics in Types and Programming Languages* pp. 45–70, MIT press.

[26] Maribel Fernandez (2004). *Programming languages and operational semantics*. Texts in Computing, Volume 1. King's college publications, London, England.

[27] John T. Boyland (2007). *Using Twelf to Prove Type Theorems*. University of Wisconsin-Milwaukee. Available at **URL**:*http://www.cs.uwm.edu/ boyland/proof/using-twelf.html*, last visited on 24/04/12.

[28] Frank Pfenning, Robert Harper and Peter Lee . *The Twelf Project*, **URL**:$http://twelf.org/wiki/Main_Page, lastvisitedon 24/04/12$.

[29] Frank Pfenning and Carsten Schürmann (1999). *System Description: Twelf A Meta-Logical Framework for Deductive Systems*. In Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction, p.202-206.