School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Computer-verified proof of type soundness for the Linear Language with Locations, L^3

by

Michael Alexander Sproul

Thesis submitted as a requirement for the degree of

Bachelor of Science (Honours)

Submitted: October 2015 Student ID: z3484357

Supervisor: Dr. Ben Lippmeier Topic ID: 3680

Computer-verified proof of type soundness for the Linear Language with Locations, L^3 Michael Alexander Sproul

Abstract

We provide a mechanically verified proof of type soundness for the previously unverified Linear Language with Locations, L^3. The language includes several low-level facilities for control over memory and *ownership*, allowing it to act as a base for the verification of systems software. L^3 is typical of modern resource-aware calculi in that it makes use of capabilities and substructural typing. Our syntactic proof of type soundness within the Coq proof assistant is a mechanisation of the existing soundness proof.

Contents

1 Introduction **1**

1.1 The Curry-Howard correspondence and the Coq proof assistant 2

1.2 Summary of Introduction 2

2 Background and Previous Work **3**

2.1 Linear and Affine Typing 4

2.2 Uniqueness Typing 6

2.3 The Linear Language with Locations, L^3 8

2.4 Systems of Capabilities 9

2.4.1 Cyclone 9

2.4.2 Pottier's type-and-capability system with hidden state 10

2.4.3 Mezzo 11

2.5 Typed assembly languages and trustworthy compilers 12

2.6 Summary of Mechanisation Techniques 13

2.7 Summary of Previous Work 13

3 Proposal **15**

3.1 Variable naming and binding 16

3.1.1 Higher-order Abstract Syntax 17

3.1.2 De Bruijn indices and the locally nameless approach 18

3.2 Proof of Type Soundness for L^3 20

3.3 Research Questions 21

3.4 Timeline 21

3.5 Summary of Proposal 22

Bibliography **23**

Chapter 1

Introduction

Computer systems form an integral part of modern society, both in the form of personal devices and critical infrastructure. Ensuring the correct operation of computer hardware and software is therefore required. One emerging technique for the construction of robust software systems is the use of mathematical formalisations and proofs of correctness. In this paradigm, desirable properties of the software can be proven true using a computer-based *proof assistant*, which itself relies on a minimal amount of trusted code. For software formalisation and verification to be truly effective, the objects under consideration must have precise mathematical models associated with them. Typically these models are created based on the *semantics* (meaning) of the programming language that the software is written in. Unfortunately for the would-be software verifier, most popular programming languages lack formal semantics and are therefore not amenable to verification techniques.

As a basis for the mechanical verification of complex languages, this thesis focuses on the mechanical verification of a small language – *The Linear Language with Locations*, L^3. Although formal semantics and semi-formal proofs of correctness exist for L^3, no computer-based proofs exist. We verify L^3 because it is typical of modern resource-aware calculi in its use of capabilities – whilst remaining simple. L^3's concepts of ownership make its mechanisation relevant to the complete verification of low-level

systems like garbage collectors and operating systems, which are at the cutting edge of verification research.

Type systems for programming languages are said to be *sound* if well-typed programs are guaranteed not to get stuck when evaluated according to the language's operational semantics. In this thesis, we aim to construct a mechanical proof of type soundness for L^3 using the Coq proof assistant. The proof will be carried out in the syntactic style of Wright and Fellesien [WF94].

1.1 The Curry-Howard correspondence and the Coq proof assistant

Recall that the Curry-Howard correspondence establishes an equivalence between logical systems and type systems for programming languages. By considering logical propositions as types, the proof of a proposition P can be given by constructing a value of type P in the equivalent programming language. The Coq proof assistant provides a dependently typed programming language with inductive data-types that allows complex propositions to be expressed and proved in this manner. Coq proofs make use of *tactics* which abstract over repetitive reasoning.

1.2 Summary of Introduction

We will prove type soundness for the Linear Language with Locations, using the Coq proof assistant. We hope that the verification of L^3 will provide insight on the tractability of mechanical verifying low-level languages with capabilities. Our broad goal is to further the construction of computer-checked proofs for all parts of the software stack.

Chapter 2

Background and Previous Work

If software verification is to propagate through modern software stacks, verification of components at different layers is a necessity. Proofs about programs written in high-level languages offer only superficial assurances if during compilation down to executable machine code the program is corrupted by an unverified transformation. Verification of entire computing systems including compilers, operating systems and file-systems is a huge undertaking however, so we restrict our attention here to the type systems of low-level languages.

Historically, low-level *systems software* has been written primarily in the C programming language, which was originally designed without a formal semantics. Attempts to assign semantics to C have resulted in numerous impressive verification projects – notably the CompCert verified compiler [Ler09] and the seL4 micro-kernel [KAE+14]. In this work we consider type systems for languages that could potentially act as verification-friendly successors to the domains where C currently excels (operating systems, language runtimes, embedded devices). Our core hypothesis is that *uniqueness types* and the destructive updates they enable should simplify the verification of systems software.

2.1 Linear and Affine Typing

Linear, affine and uniqueness typing are closely-related features of type systems that enforce rules about the number of times values may be used and referenced. These restrictions are motivated by several desirable properties that can be obtained by enforcing them. The Clean programming language (introduced in [BvEvLP87]) uses uniqueness typing to ensure that values in memory have at most one reference to them, thus enabling *destructive updates* whilst preserving referential transparency. The Rust programming language [Moz15] uses uniqueness typing to track and free heap-allocated memory, thus allowing it to achieve memory safety without garbage collection. This makes it suitable for writing systems software where a garbage collector isn't available, like a garbage collector itself, or an operating system.

In classical and intuitionistic logic there are structural deduction rules equivalent to the following, called *Contraction* and *Weakening* [Wad90, Wad93].

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction} \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening}$$

Contraction allows duplicate assumptions to be discarded, whilst Weakening allows a non-vital extra assumption to be introduced from nowhere. Linear logic [Gir87] selectively restricts the use of both rules, such that every *linear* assumption is used *exactly once*. Affine logic on the other hand restricts only the use of Contraction, resulting in each linear assumption being used *at most once*. Both are *substructural* in that they restrict the use of structural rules.

Non-linear assumptions can still be used an unlimited number of times. Hence, variants of Contraction and Weakening that operate only on non-linear assumptions are present in linear logic, affine logic and similar substructural systems.

When transformed into typing rules (as per Section 1.1), Contraction and Weakening take the form:

$$\frac{\Gamma, y : A, z : A \vdash u :: B}{\Gamma, x : A \vdash u[x/y, x/z] :: B}\ \text{Contraction} \qquad \frac{\Gamma \vdash y :: B}{\Gamma, x : A \vdash y :: B}\ \text{Weakening}$$

Note that the use of substitutions in the rule for Contraction is required so that no variable appears more than once in a typing context Γ. Further, substructural type theories include context-splitting operations that allow the linear variables of a typing context to be split between two new contexts which can be used to type-check subterms.

As in linear and affine logic, linear type theory restricts the use of both Contraction and Weakening, whilst affine type theory restricts just the use of Contraction. Our previous observations about the number of times an assumption is used, now translate into observations about the number of times variables are used.

Without Contraction, *variables can only appear once in a term.* The dual substitution of x for y and x for z allows a term containing one y and one z to become a term containing two occurrences of x. None of the other rules of intuitionistic type theory allow this [Wad93].

Similarly, without Weakening, *all variables in the context must be used in the term.* This follows from the fact that Weakening introduces an unused variable to the context and no other rule of intuitionistic type theory allows this [Wad93].

From these two observations we can conclude that linear type theory requires all variables to be used *exactly once* in terms, whilst affine type theory requires all variables to be used *at most once* in terms.

As in substructural logics, substructural type theories selectively permit Contraction and Weakening for select non-linear values. This is quite practical in the context of programming language implementation as it allows trivially copyable values like integers to be easily duplicated. The mechanism employed by Wadler [Wad93] involves differentiating between linear and non-linear assumptions, and adding type and value-level bang (!) operators to make types and values non-linear. The function for duplication of non-linear values (of type $!A$) is expressed as follows in Wadler's system:

$$\varnothing \vdash \lambda\langle x' \rangle.\ \text{case}\ x' \text{of}\ !x \to \langle !x, !x \rangle :\ !A \multimap (!A \otimes !A)$$

See [Wad93] for full notational details.

2.2 Uniqueness Typing

Although linear and affine type theory capture the essence of uniquely referenced values, they are insufficient to describe the concept of *uniqueness* as it appears in languages like Clean. In his 2007 paper, de Vries [dVPA07] notes that terms of a unique type should be *guaranteed to never have been shared*, which is sufficient to guarantee a unique pointer at runtime. In contrast, terms of linear (or affine) type are *guaranteed not to be shared in the future*, which is insufficient to guarantee a unique pointer.

The distinctness of linearity and uniqueness is further highlighted by the *dereliction* rule present in some versions of linear type theory, and the rule that we'll refer to as *uniqueness removal* present in Clean's type system. Let $\alpha^{\odot}$ and $\alpha^{\otimes}$ stand for linear and non-linear versions of a base type α. Similarly, let $\alpha^{\bullet}$ and $\alpha^{\times}$ stand for unique and shared versions of a base type α. We have:

$$(\lambda x.\ x)\ :\ \alpha^{\otimes} \to \alpha^{\odot} \quad \text{(linear dereliction)}$$
$$(\lambda x.\ x)\ :\ \alpha^{\bullet} \to \alpha^{\times} \quad \text{(uniqueness removal)}$$

The dereliction rule allows a non-linear value to be transformed into a linear one. As noted by de Vries [dV08], an analogous rule for unique types that converts shared values to unique ones cannot possibly be sound. The "unique" value resulting from such a rule would not necessarily be unique because other shared references may still exist.

Conversely, uniqueness removal only makes sense in the context of uniqueness types. A unique value may sacrifice its uniqueness to become shared, but a linear value which

models the existence of a single resource should not be transformed into an unlimited supply of that resource.

Note that dereliction need not form a part of type systems based on linear logic, and that it is absent from Wadler's presentations [Wad90, Wad93] and all systems considered in the next sections. Further note that if uniqueness is to be exploited to make garbage collection unnecessary – as in the case of Rust – then the uniqueness removal rule is undesirable as it prevents values from having a unique owner.

Due to the non-equivalence of linearity and uniqueness, de Vries constructed a distinct set of semantics and typing rules to model Clean's type system [dVPA07].

One key component of his approach is the use of the *kind* (type of types) system to track uniqueness and non-uniqueness. As in Haskell, de Vries' uniqueness system includes a kind for data ($*$) which is the kind of all inhabited types (and `Void`). In addition, there is a uniqueness kind $\mathcal{U}$ inhabited by two types $\bullet$ and $\times$ representing uniqueness and non-uniqueness respectively. A third kind, $\mathcal{T}$ is the kind of base types (like `Int`). These kinds are brought together by a type constructor $\texttt{Attr} ::_k \mathcal{T} \to \mathcal{U} \to *$ which applies a uniqueness attribute to a base type to form a type that is inhabited. For example, $\texttt{Attr} \bullet \texttt{Int}$ or $\texttt{Int}^\bullet$ is the type of uniquely referenced integers.

The other main technique employed by de Vries' model is the use of arbitrary boolean expressions as uniqueness attributes, with $\bullet$ as true and $\times$ as false. Clean's type system allows uniqueness polymorphism, which results in constraint relationships between uniqueness variables, which are represented in de Vries' system as simple boolean expressions that can be handled by a standard unification algorithm.

Importantly, de Vries' work includes the only known mechanical verification of a type system similar to Clean's. The formalisation uses the Coq proof assistant, and the *locally nameless* approach to variable naming that is discussed in Section 3.1.2.

2.3 The Linear Language with Locations, L^3

The Linear Language with Locations L^3 [MAF05], makes use of concepts from linear typing to manage *capabilities* – values that represent the permission to read or write an area of memory. By treating capabilities linearly, pointers themselves become duplicable and can be stored in numerous locations freely.

L^3 is a *low-level* language by design and features primitive operations for allocating and deallocating memory. By virtue of linear capabilities, *strong updates* are supported, whereby the type of a value in a memory location may change upon writing. Strong updates enable staged value initialisation, and simulation of register type-changes throughout program execution. For example, a pointer initially pointing to an `Int` can be overwritten with a `Bool` and the type system can statically track this change.

L^3's definition includes a description of syntax, operational semantics and typing rules. Notably, the operational semantics include a store type which maps locations to values. Similarly, the typing rules include a location context that tracks which locations are in scope.

$$(\sigma, \text{new } v) \Rightarrow (\sigma \uplus \{l \mapsto v\}, \ulcorner l, \langle \text{cap}, \text{ptr } l \rangle \urcorner)$$

$$(\sigma, \text{let } \ulcorner \rho, x \urcorner = \ulcorner l, v \urcorner \text{ in } e) \Rightarrow (\sigma, e[l/\rho][v/x])$$

These two rules from the operational semantics show the behaviour of the `new` keyword for allocating memory, and the `let` construct for unpacking pointers and capabilities. Note how the store σ is extended with a mapping from a new location l to the value v in the case of the rule for `new`. The $\ulcorner l, \langle \text{cap}, \text{ptr } l \rangle \urcorner$ notation represents a value of *existential type* that witnesses the existence of a pair containing a capability for, and pointer to, some location l which is hidden from the programmer.

L^3's rules for managing capabilities are closer to standard linear typing than de Vries'

rules for modelling Clean's uniqueness typing. Typing assumptions are treated linearly, there are context-splitting operations and contraction and weakening are permitted for values of bang (!) type. This is in contrast to the use of kinds and attribute type constructors in de Vries' model.

Conceptually, because the rules that introduce capabilities are baked into the type system, capabilities are guaranteed *not to have been shared* (uniqueness) and *not to be shared in the future* (linearity). The use of capabilities also yields all of the same properties yielded by substructural and uniqueness typing – notably destructive updates and the potential to eliminate garbage collection. It is typical of many of the later systems which we cover in the next section.

2.4 Systems of Capabilities

2.4.1 Cyclone

Several systems extend and generalise the capability-based approach employed in L^3. Fluet, Morrisett and Ahmed followed up their paper on L^3 with a region-based system that borrows many ideas from L^3, called λ^{rgnUL} [FMA06]. Notably, it makes use of linear capabilities to provide safe access to *dynamic regions*, which are first-class abstractions for the allocation of memory. Dynamic regions extend simpler lexical regions by allowing regions to exist independent of lexical scopes. Accompanying the λ^{rgnUL} paper is a mechanised proof of type soundness using the Twelf proof assistant [PS99].

The same authors are also responsible for the Cyclone project [GHJM05], which extends the C programming language with regions and uniqueness typing in order to achieve safe memory management without garbage collection or manual intervention. The λ^{rgnUL} calculus models Cyclone's core features, and there exists a translation from Cyclone to λ^{rgnUL} via an intermediate language F^{RGN} which makes use of a generalised ST monad [FMA06, FM04]. No mechanised proofs of correctness for this work

exist, although an earlier semi-formal proof of type soundness for Cyclone [JMG+01] is structured in a way that looks amenable to mechanised verification. On their webpage [GHJ+15], the creators of Cyclone note that work on the project has stopped, with many of the ideas living on in Rust. Future formalisations of Rust can hopefully make use of this work.

2.4.2 Pottier's type-and-capability system with hidden state

A mechanical formalisation for a system even more similar to L^3 than λ^{rgnUL} is given in a 2013 article by François Pottier [Pot13a]. Pottier's system, SSPHS , uses affine capabilities in the style of L^3, but adds polymorphism and support for *hidden state*. Hidden state allows an object to completely conceal mutable internal state from its clients. Pottier gives a memory manager as an example where such a feature is useful – clients care only about the memory allocated or de-allocated, and not about internal data-structures modified in the process. Hidden state is realised via a typing rule called the *anti-frame rule*, which makes terms with hidden state subtypes of the type sans hidden state.

The concept of hidden state is distinct from, yet related to, the existential types that L^3 uses to conceal exact locations. SSPHS also employs hidden state for the purpose of general resource management, rather than just memory management. The ability to express memory management in the language obsoletes L^3 and similar systems' explicit rules for memory management, which Pottier describes as "magic" [Pot13a].

All of L^3's features, including strong updates, are covered by Pottier's system. It also subsumes λ^{rgnUL} , with support for polymorphism and regions. Unlike previous systems it also guarantees the runtime-irrelevance of capabilities, which are proved to be erasable.

Pottier's formalisation is done within the Coq proof assistant and makes use of de Bruijn indices for variable binding (a pre-cursor to his `dblib` library, discussed in Section 3.1.2). The formalisation consists of 20,000 lines of Coq source and follows the

syntactic approach to proving type soundness via progress and preservation. Pottier notes that the formalisation took around 6 months to complete.

2.4.3 Mezzo

Together with Thibaut Balabonski and Jonathan Protzenko, Pottier is also responsible for the Mezzo programming language and its associated Coq formalisation [BPP14]. Mezzo differs from SSPHS and λ^{rgnUL} in that it is designed to be high-level and expressive. Like the other systems examined, its system of ownership is based around linear *permissions*, which allow programmers to design diverse usage *protocols* for functions and data. Mezzo's model of concurrency leverages ownership to guarantee that well-typed programs do not contain data-races, a property that is also formalised in Coq.

Mezzo includes mechanisms for deferring permissions checks to runtime in order to gain more expressive power, at the cost of some synchronisation overhead. Its surface syntax is also designed to be more minimal than languages like L^3 which favour explicit annotations. Both of these aspects reflect Mezzo's ambition to be a user-facing programming language that provides control over resources.

The prototypical compiler for Mezzo uses untyped OCaml as its target language and as such requires garbage collection at runtime. Further, due to OCaml's lack of parallelism, concurrent and race-free Mezzo programs are currently unable to take advantage of multiple cores. One can imagine further work to compile Mezzo to a low-level language with similar semantics, in order to take advantage of its full feature set.

Mezzo's Coq formalisation consists of 14,000 lines of code and makes use of a 2000 line library called `dblib` for handling de Bruijn indices. Like the proof for SSPHS , it uses progress and preservation to prove type soundness.

2.5 Typed assembly languages and trustworthy compilers

Strong updates can be used to model the storage of type-distinct values in a single register through-out program execution. As such, low-level calculi like L^3 and λ^{rgnUL} are conceptually linked to *typed assembly languages* (TALs), which extend regular assembly languages with type annotations.

Well-typed TAL programs typically guarantee memory safety given an axiomatisation of a machine architecture. In the TALx86 [MWCG99, CGG+99] system, blocks are annotated with pre-conditions that place requirements on the types of registers. This approach to typing is substantially different from the operational semantics and inductive typing judgements used to describe the semantics of the other languages we've surveyed (L^3, λ^{rgnUL} , SSPHS). However, recent work by Amal Ahmed *et al.* has successfully resulted in a more traditional model for typed assembly languages [AAR+10]. This model still differs from the others considered in this thesis in that it uses denotational semantics, Hoare logic and several interconnecting layers in order to minimise the number of axioms required. Ahmed's paper includes a Twelf formalisation of soundness for the TAL semantic framework and an example language.

Another take on the typed assembly language concept, is Bedrock from Adam Chlipala's research group [Chl11]. Bedrock uses a domain-specific assembly language embedded within Coq to express low-level programs. Aided by user-provided annotations, Bedrock can prove properties about these assembly programs in an automated way using custom Coq tactics. The block annotations resemble the block pre-conditions of TALx86.

More broadly it is worth noting the contribution of the CompCert [Ler09] project to program verification. Through a series of semantics-preserving translations through intermediate languages, CompCert compiles a variant of C to multiple assembly languages. CompCert is programmed and verified in Coq. Verification of programs written in a low-level linearly-typed language could use parts of CompCert, perhaps with a language like L^3 or SSPHS as an intermediate language.

2.6 Summary of Mechanisation Techniques

The following table contains a summary of languages and type systems and their mechanisations. A tick (✓) indicates that a property is true for a given language, a cross (×) indicates that it is false and a dash (-) indicates that the property is not applicable. The * indicates work to be completed as part of this thesis. Note that we also write "Clean" here to mean Edsko de Vries' uniqueness typing system [dVPA07].

System	**DU**	**SU**	**Cp**	**Poly**	**Other**	**Mechanised?**	**Naming**
Clean [dVPA07]	✓	×	×	✓	-	✓(Coq)	LN
Rust [Moz15]	✓	×	-	✓	No GC	×	-
L^3 [MAF05]	✓	✓	✓	×	-	Yes* (Coq)	DB*
λ^{rgnUL} [FMA06]	✓	-	✓	✓	Cyclone base	✓(Twelf)	HOAS
SSPHS [Pot13a]	✓	✓	✓	✓	Hidden state	✓(Coq)	DB
Mezzo [BPP14]	✓	✓	✓	✓	Data-race free	✓(Coq)	DB

Key: DU=Destructive Updates, SU=Strong Updates, Cp=Capabilities, Poly=Polymorphism, LN=Locally Nameless, DB=De Bruijn Indices, HOAS=Higher-order Abstract Syntax.

2.7 Summary of Previous Work

In summary, previous work on the formalisation of resource-aware type systems has culminated in the wide-spread use of capabilities. The basic ideas of linear and affine logic have been adapted to form the core of these systems, with some extra features and approaches mixed in (e.g. hidden state and de Vries' use of kinds). The use of mechanical verification in proofs of type soundness has gained popularity, with most recent works including a formalisation in Coq or Twelf. Other mainstream proof assistants like Isabelle seem to be less used in this space, but we suspect this is primarily due to the limited number of research groups performing this kind of research, and their personal preferences. The notable exception to the verification trend is the L^3 language,

which remains unverified but has had all of its salient features mechanically verified as part of other projects. Several capability systems mention Alias Types [SWM00] and separation logic [Rey02] as foundational concepts, but we defer discussion of these to future work.

Chapter 3

Proposal

The aim of this thesis is to formalise the semantics of the Linear Language with Locations (Core L^3) using the Coq proof assistant. The first step will be to translate the operational semantics and typing rules from the paper [MAF05] into inductive Coq definitions. As part of this, we will have to define two substitution functions – one for the term substitution and one for location substitution. We will use the `dblib` library [Pot13b] for both substitution functions. The justification for this choice and a summary of the associated problems and alternatives is given in Section 3.1.2.

With the operational semantics and typing rules defined, we will proceed with a syntactic proof of type soundness via progress and preservation. Experience with simpler languages from *Software Foundations* [PCG+15] suggests that the proof of progress will be less involved than the proof of preservation. The choice to aim for a syntactic proof is motivated by the semi-formal syntactic proof given in the L^3 paper, and the efficacy of syntactic soundness proofs for languages surveyed in the literature review. Considerations of how the proof will be undertaken are given in Section 3.2.

Time permitting, the work will conclude with proofs of other properties of L^3. One interesting proof may be of the runtime irrelevance of capabilities. As in Pottier's work on SSPHS [Pot13a] this could be done by defining a version of the operational semantics in which capabilities do not appear, and proving that the two sets of semantics are

equivalent.

Section 3.3 includes research questions that this thesis hopes to address, whilst Section 3.4 gives an approximate timeline for the completion of the work.

3.1 Variable naming and binding

One problem that arises frequently in the formalisation of language semantics is that of *capture-avoiding substitution*. Substitution operations, whereby a value is substituted for a variable in a term, form the core computational component of the operational semantics in many languages. In the simply-typed (and untyped) lambda calculus, the β-rule uses substitution (denoted $[v/x]e$) to describe the semantics of function application:

$$(\lambda x : \tau.\ e)\ v \Longrightarrow_\beta [v/x]e$$

The problem of *variable capture*, which we wish to avoid, is demonstrated by the following example:

$$(\lambda x.\ \lambda y.\ x + y)\ y \not\Longrightarrow_\beta (\lambda y.\ y + y)$$

Here the parameter y is a free variable acting as a place-holder for a value in the environment. After substitution however, the y replacing x in the abstraction body $x + y$ becomes bound due to the name collision between the free y and the binder y. This altering of the meaning of terms during substitution is something we would like to avoid.

One way to avoid variable capture is to forbid the substitution of any terms containing free variables. In such a system, free variables like y are never considered values and as

such cannot be used in variable capturing substitutions. This is the approach taken by *Software Foundations* [PCG+15] in formalisations of the simply-typed lambda calculus and its variants. A further consequence of this approach is that globally-shared integers or strings for variable names are sufficient to guarantee soundness. Although it's tempting to embrace this approach for its simplifying properties, the L^3 specification requires *"standard capture-avoiding substitution"* [MAF05].

In our Coq formalisation of L^3 we would therefore like to include capture-avoiding substitution as part of the definitions of variable names and substitution operations. For this we consider three main approaches from the literature which all exploit the observation that the exact names of bound variables are insignificant at the level of language formalisation. In other words, although the names of variables may hold meaning for the authors of programs, they do not impact the meanings of programs themselves.

3.1.1 Higher-order Abstract Syntax

When using Higher-order Abstract Syntax (HOAS) to handle variable binding, the binders of the host language (in our case Coq) are used to represent binding constructs in the object language. Twelf encourages use of HOAS through its light-weight syntax (this example adapted from [Twe08]):

```
exp : type.
let : exp -> (exp -> exp) -> exp.
```

The full definition for `exp` is omitted, but this example demonstrates that a let-binding in the *object language*, can be considered in the *meta-language* as a value representing the expression being bound, and a function that accepts that bound expression as input. For example, the object language expression `let x = 1 + 2 in x + 3` would be encoded as `let (plus 1 2) ([x] plus x 3)`, where `plus : nat -> nat -> expr` and `([x] e)` is syntax for $(\lambda x.\ e)$.

This sort of encoding becomes problematic in Coq due to the difficulty of encoding types involving *negative occurrences* inductively. A type appears as a negative occurrence if it would appear below an odd number of negations in a translation to classical logic [Pie02]. In our example, the argument to `let`'s higher-order function is a negative occurrence: `exp -> (exp -> exp) -> exp`. An (invalid) inductive Coq definition for the above Twelf example would be:

```
Inductive exp : Set :=
  | exp_plus : nat -> nat -> exp
  | exp_let : exp -> (exp -> exp) -> exp.
```

Coq rejects this definition with the error: `Non strictly positive occurrence of "exp" in "exp -> (exp -> exp) -> exp"`, as expected.

There are ways to simulate HOAS-like systems in Coq by either limiting the expressiveness and defining filters on the inductive types obtained [DFH95] or by mixing de Bruijn indices and HOAS [CF07]. As HOAS is entirely absent from the Coq formalisations surveyed we choose to look past it in favour of plain de Bruijn indices.

3.1.2 De Bruijn indices and the locally nameless approach

Building on the idea that the exact names of bound variables are irrelevant, de Bruijn indices represent variables as *distances from their binding occurrence* [DB72]. For example, the identity function $(\lambda x.\ x)$ is encoded as $(\lambda.\ 0)$ – assuming without loss of generality that there are no integer literals in the object language that could be confused for de Bruijn indices.

For terms that contain free variables, a fixed naming context is used to map free variables to indices [Pie02]. For example, with the naming context $\Gamma = x, y, z$ which maps $\{x \mapsto 2, y \mapsto 1, z \mapsto 0\}$, the term $(\lambda x.\ (x\ y)\ z)$ would be encoded as $(\lambda.\ (0\ 2)\ 1)$. We can imagine the context prepended to the term as an ordered list of binders, so that the use of z ends up being separated from its binding occurrence by *1* – the lambda.

Capture-avoiding substitution with de Bruijn indices can be defined as a recursive function that makes use of a *shifting* operation. Shifting a term by d conceptually renumbers free variables for the introduction of d elements at the end of the naming context. To avoid renumbering bound variables, a cut-off parameter c is threaded through the computation. We denote shifting a term t by d with cut-off c as $\uparrow_c^d t$.

$$\begin{aligned} \uparrow_c^d k &= \begin{cases} k & \text{if} \quad k < c \\ k + d & \text{if} \quad k \geq c \end{cases} \\ \uparrow_c^d (\lambda.\ t_1) &= \lambda.\ \uparrow_{c+1}^d t_1 \\ \uparrow_c^d (t_1\ t_2) &= (\uparrow_c^d t_1)\ (\uparrow_c^d t_2) \end{aligned}$$

With shifting defined, the definition of substitution is straight-forward – we simply shift the free variables of the substituted term each time we move under a lambda.

$$\begin{aligned} [s/j]k &= \begin{cases} s & \text{if} \quad j = k \\ k & \text{otherwise} \end{cases} \\ [s/j]\,(\lambda.\ t_1) &= \lambda.\ [(\uparrow_0^1 s)/(j+1)]\ t_1 \\ [s/j]\,(t_1\ t_2) &= ([s/j]\ t_1)\ ([s/j]\ t_2) \end{aligned}$$

These equations for shifting and substitution are due to [Pie02].

Unlike HOAS, the recursive functions for de Bruijn indices are well-suited for use with the Coq proof assistant. Of the Coq formalisations surveyed in our literature review, two of the largest and most similar to our planned formalisation use de Bruijn indices. The first, SSPHS [Pot13a] defines a module with several lemmas about substitution, while the Mezzo formalisation [BPP14] makes use of a stand-alone library called `dblib` [Pot13b]. This library uses Coq's type-classes to provide useful substitution lemmas, given the definitions of a few basic operations on terms of the object language. This is

an appealing prospect, and we hope that by using `dblib` for our formalisation we will be able to assess its suitability as a generic library for de Bruijn indices.

The alternative to `dblib` would have been to use Arthur Charguéraud's *Engineering Formal Metatheory* library [ACP+08] for binding using a *locally nameless* (LN) representation. The locally nameless representation uses de Bruijn indices for bound variables and traditional names for free variables. In his formalisation of uniqueness typing Edsko de Vries notes that use of the LN library *"meant that little of our subject reduction proof needs to be concerned with alpha-equivalence or freshness"* [dVPA07].

However, LN does depend on Charguéraud's TLC library for *non-constructive* logic within Coq [Cha15]. We elect not to use this library, in order to keep the set of axioms minimal and to aid compatibility with other proofs. Further, the formalisations of SSPHS and Mezzo make successful use of de Bruijn indices and they are considerably closer to L^3 than de Vries' uniqueness typing system.

3.2 Proof of Type Soundness for L^3

To aid in the syntactic proof of type soundness for L^3 several sources will be drawn upon for inspiration. The paper proof of soundness for L^3 should provide some hints of lemmas to prove and techniques for proving them, like which variable to do induction on. The paper proof of soundness for core L^3 spans only 8 pages, so the work could always expand to verify the extended version of L^3 which requires a further 14 pages of paper proof.

Other sources of inspiration will be the proofs of soundness for Mezzo and SSPHS – particularly for Coq-specific verification techniques. The Twelf proof of soundness for λ^{rgnUL} might also be useful for more general techniques.

3.3 Research Questions

By conducting the above research, we hope to answer the following research questions:

- Does the L^3 type system admit straight-forward verification in the style of λ^{rgnUL} , Mezzo and other modern capability systems, or do the generalisations and simplifications or these systems make verification simpler?
- Is the `dblib` library for de Bruijn indices general enough to allow the verification of languages it wasn't designed for? Specifically, is it possible to define term and location substitution for L^3 using an unmodified version of `dblib`?
- How much effort is involved in the translation of syntactic soundness proofs from paper to the Coq proof assistant?

To quantify the effort involved in the proofs, we will make use of a simple script that records every minute spent inside the Coq IDE.

3.4 Timeline

I plan to complete most of the practical work over the summer so that the thesis write-up can be completed in Semester 1 2016. In case of delays, semester 1 can also act as an emergency buffer.

Over summer (3 months ≈ 12 weeks total):

- L^3's operational semantics and type system in Coq (2 weeks).
- Prove progress (4 weeks).
- Prove preservation, and related lemmas (6 weeks).

We expect that some definitions will need to be adjusted along the way.

Semester 1 2016 (12 weeks):

- Finalise proofs completed over the summer break (6 weeks).
- Write-up results and findings (6 weeks).

Time permitting, the previously discussed extra proofs could be undertaken either during summer, or after the completion of the main part of the thesis write-up.

3.5 Summary of Proposal

By drawing inspiration from the paper proof of soundness for L^3, and similar proofs from the literature, we will prove type soundness for Core L^3. We will use the syntactic soundness technique first defined by Wright and Felleisen [WF94]. We will make use of the `dblib` library [Pot13b] for reasoning about de Bruijn indices. Time permitting, proofs of soundness for extended L^3, or of other properties of Core L^3 may be attempted.

Bibliography

[AAR+10] Amal Ahmed, Andrew W Appel, Christopher D Richards, Kedar N Swadi, Gang Tan, and Daniel C Wang. Semantic foundations for typed assembly languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):7, 2010. pages 12

[ACP+08] Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN Notices*, volume 43, pages 3–15. ACM, 2008. pages 20

[BPP14] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. *Submitted for publication*, 2014. pages 11, 13, 19

[BvEvLP87] T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean — a language for functional graph rewriting. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer Berlin Heidelberg, 1987. pages 4

[CF07] Venanzio Capretta and Amy P Felty. Combining de bruijn indices and higher-order abstract syntax in coq. In *Types for Proofs and Programs*, pages 63–77. Springer, 2007. pages 18

[CGG+99] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35, 1999. pages 12

[Cha15] Charguéraud, Arthur. TLC: a non-constructive library for Coq. http://www.charglueraud.org/softs/tlc/, 2015. pages 20

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *ACM SIGPLAN Notices*, 46(6):234–245, 2011. pages 12

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the

church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972. pages 18

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. *Higher-order abstract syntax in Coq.* Springer, 1995. pages 18

[dV08] Edkso de Vries. *Making Uniqueness Typing Less Unique.* PhD thesis, Trinity College Dublin, 12 2008. pages 6

[dVPA07] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 201–218, 2007. pages 6, 7, 13, 20

[FM04] Matthew Fluet and Greg Morrisett. Monadic regions. In *ACM SIGPLAN Notices*, volume 39, pages 103–114. ACM, 2004. pages 9

[FMA06] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 7–21. Springer Berlin Heidelberg, 2006. pages 9, 13

[GHJ+15] Dan Grossman, Michael Hicks, Trevor Jim, Greg Morrisett, and Nikhil Swamy. Cyclone the language, 2015. pages 10

[GHJM05] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of c. *C/C++ Users Journal*, 23(1):112–139, 2005. pages 9

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. pages 4

[JMG+01] Trevor Jim, Greg Morrisett, Dan Grossman, Yanling Wang, James Cheney, and Mike Hicks. Formal type soundness for cyclone's region system. Technical report, Cornell University, 2001. pages 10

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014. pages 3

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. pages 3, 12

[MAF05] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A Linear Language with Locations. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 293–307. Springer Berlin Heidelberg, 2005. pages 8, 13, 15, 17

[Moz15] Mozilla Resarch. The Rust Programming Language. https://rust-lang.org/, 2015. pages 4, 13

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999. pages 12

[PCG+15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015. http://www.cis.upenn.edu/~bcpierce/sf. pages 15, 17

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. pages 18, 19

[Pot13a] François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of functional programming*, 23(01):38–144, 2013. pages 10, 13, 15, 19

[Pot13b] François Pottier. The `dblib` library for de Bruijn indices in Coq. https://github.com/fpottier/dblib, 2013. pages 15, 19, 22

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16*, pages 202–206. Springer, 1999. pages 9

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002. pages 14

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Programming Languages and Systems*, pages 366–381. Springer, 2000. pages 14

[Twe08] Twelf Project. Twelf Wiki - Higher-order abstract syntax. http://twelf.org/wiki/Higher-order_abstract_syntax, 2008. pages 17

[Wad90] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990. pages 4, 7

[Wad93] Philip Wadler. A taste of linear logic. In AndrzejM. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 185–210. Springer Berlin Heidelberg, 1993. pages 4, 5, 6, 7

[WF94] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994. pages 2, 22