School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

A Library Based Approach to the Verification of Languages with Linear Types

by

Michael Alexander Sproul

Thesis submitted as a requirement for the degree of

Bachelor of Science (Honours)

Submitted:	May 2016	Student ID:	z3484357
Supervisor:	Dr. Ben Lippmeier	Topic ID:	3680

A Library Based Approach to the Verification of Languages with Linear Types Michael Alexander Sproul

Abstract

We present a library of proofs to aid in the mechanical verification of languages with substructural type systems. We identify *context splitting* as a widely applicable technique in the verification of these languages, and argue that our proofs are relevant to any language that uses multiple contexts to separate linear assumptions from non-linear ones, in the style of Dual Intuitionistic Linear Logic. Our work is implemented using the Coq interactive theorem prover and builds upon François Pottier's DbLib library for variable binding using De Bruijn indices. We provide a proof of type soundness for a linear lambda calculus and highlight how similar techniques might be applied to *The Linear Language with Locations*, L^3 – which is representative of a class of more complex languages with destructive updates.

Acknowledgements

I would like to thank my supervisor, Ben Lippmeier, for his sage advice and encouragement. I would also like to thank the awesome Programming Languages and Systems research group at UNSW for stimulating my imagination and supporting my project. In particular, I'm grateful to Gabi and Manuel for their leadership and for making it all possible. I would also like to thank my family, friends and Lily for supporting me throughout and keeping me grounded. Thank you.

Contents

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 a worthwhile endeavour. 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.

The C Programming Language, in which large amounts of low-level *systems software* is written, is an example of a language that is difficult to reason about because of its murky semantics. According to the language specification the results of some operations are classified as *undefined behaviour*, meaning that the compiler has no obligation to produce a specific result [ISO11]. Ideally, we would like to rule out undefined behaviour at compile time through the use of a type system. In this thesis we focus on the formal

semantics of languages with strong type systems that are more well-suited to low-level software verification than C.

In particular, we are interested in *resource-aware* languages which exploit *substructural* type systems based on linear logic [Gir87] to track how values are consumed and destroyed. Intuitively, a *linear* value is one that is guaranteed to be used exactly once. In the context of systems software, we can use variants of linearity to enable destructive updates for uniquely reference values, or automatic memory management without garbage collection.

We begin by discussing the linear lambda calculus (LLC), which is a simple extension of λ-calculus with a linear type system. We then go on to describe more expressive languages which include features like shared pointers and destructive updates.

The **main contribution** of this thesis is a software library for the interactive theorem prover Coq. The library includes definitions and lemmas primarily about *context splitting*, which is a technique used in many substructural type systems. Our code is an extension of François Pottier's DbLib library which aims to de-duplicate the effort required to reason about variable binding, just as we hope to de-duplicate the effort required to reason about linear typing.

As a proof-of-concept for our approach we have undertaken a proof of *type soundness* for a linear lambda calculus. Type soundness is a key result in establishing the sensibility of a language and we argue that our approach is also applicable to soundness proofs for more complex languages with the same foundations. Specifically, we argue that a proof of soundness for *The Linear Language with Locations* (L^3) might make effective use of our library, and that L^3 is representative of languages that build upon the linear lambda calculus by adding destructive updates and shared pointers.

Chapter 2

Background

2.1 Coq and the Curry-Howard Correspondence

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.

2.2 Beyond C

Historically, low-level *systems software* has been written primarily in the C programming language, which was originally designed without a formal semantics. Despite this, a large body of work has developed around semantics for C and the verification of low-level software that such work enables. Michael Norrish's contribution of a structural operational semantics for C [Nor98] is notable in that it laid the foundations for C verification projects like the seL4 micro-kernel [KAE+14]. The seL4 kernel is written

in C and is accompanied by proofs of correctness about its security, including crash-freedom and the absence of memory safety violations. The kernel makes use of the AutoCorres tool for translating C code into a higher-level monadic language embedded in the Isabelle/HOL theorem prover [GLAK14]. AutoCorres generates proofs that the translation is sound with respect to the semantics of the source C program. Another notable C verification project is the CompCert verified compiler by Xavier Leroy [Ler09], which is capable of compiling and optimising C code to assembly whilst preserving its semantics. However, CompCert does not guarantee the absence of undefined behaviour in compiled programs, and can only detect undefined behaviour in code that doesn't perform I/O, by dynamically interpreting it.

In this thesis we are concerned with the formal foundations for a possible successor to C. Whether or not this approach will result in simpler and less labour-intensive software verification is an open question. However, we hope that by starting afresh, and using types to enforce useful invariants, a new language for low-level software verification can emerge.

2.3 Overview of 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.

At their core, all of these systems enforce their constraints using typing rules derived

from linear logic [Gir87]. To get a feel for linear logic, we first turn our attention to the linear lambda calculus. We then continue by discussing more complex languages and the techniques used in their mechanical formalisation.

2.4 The Linear Lambda Calculus

For our presentation of the linear lambda calculus we follow the work of Plotkin and Barber [Bar96] on Dual Intuitionistic Linear Logic (DILL). First, the syntax for variables, types and terms:

$$\begin{array}{rcl} x, y & \in & \textbf{Vars} \\ A, B & ::= & \mathbb{I} \mid A \otimes B \mid A \multimap B \mid !A \\ t, u & ::= & x \mid * \mid \textbf{let } * \textbf{ be } t \textbf{ in } u \mid t \otimes u \mid \textbf{let } x \otimes y : A \otimes B \textbf{ be } t \textbf{ in } u \mid \\ & & \lambda x : A.\ t \mid (t\ u) \mid\ !t \mid \textbf{let } !x : A \textbf{ be } t \textbf{ in } u \end{array}$$

The meta-variables x and y range over a countably infinite set of variables, **Vars**. Meta-variables A and B range over types, and t and u range over terms. The core constructs of the language are those of the lambda calculus: variables (x), abstractions ($\lambda x : A.\ t$) and applications ($t\ u$). We introduce the other constructs through the typing rules of the language.

We write the judgement $\Gamma; \Delta \vdash t : A$ to denote that a term t has type A relative to two typing environments Γ and Δ which record the types of free variables in t. The two typing environments are the origin of the name **Dual** Intuitionistic Linear Logic, and are part of DILL's mechanism for differentiating linear and non-linear values.

Typing environments can have many representations, and for the purposes of precise mechanical formalisation this will become very important (see §3.5). In "semi-formal" contexts, including Barber's presentation of DILL, the environments are defined to be sequences of variable-type pairs $x_0 : A_0, x_1 : A_1, \ldots$ such that no variable appears more than once. Comma-separated environments like $(\Gamma, x : A)$ and (Δ_1, Δ_2) are understood to be the concatenation of two environments with disjoint sets of variables.

The first environment, Γ, records the types of non-linear intuitionistic variables. In contrast, the second environment Δ records the types of linear variables which must be used exactly once. Although a language with *just* linear values is easily conceivable, it is both more practical and flexible to allow non-linear values to exist alongside linear ones. Trivially copyable values like integers are well-suited to being non-linear in a linear language.

To define which typing judgements are well-formed, we use inference rules in the style of natural deduction:

$$\frac{}{\Gamma, x : A; \varnothing \vdash x : A}\ \text{(Int-Var)} \qquad \frac{}{\Gamma; x : A \vdash x : A}\ \text{(Lin-Var)}$$

$$\frac{}{\Gamma; \varnothing \vdash * : \mathbb{I}}\ \text{(Unit-I)} \qquad \frac{\Gamma; \Delta_1 \vdash t : \mathbb{I} \quad \Gamma; \Delta_2 \vdash u : A}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let } * \textbf{ be } t \textbf{ in } u : A}\ \text{(Unit-E)}$$

$$\frac{\Gamma; \Delta_1 \vdash t : A \quad \Gamma; \Delta_2 \vdash u : B}{\Gamma; \Delta_1, \Delta_2 \vdash t \otimes u : A \otimes B}\ (\otimes\text{-I}) \qquad \frac{\Gamma; \Delta_1 \vdash u : A \otimes B \quad \Gamma; \Delta_2, x : A, y : B \vdash t : C}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let } x \otimes y : A \otimes B \textbf{ be } u \textbf{ in } t : C}\ (\otimes\text{-E})$$

$$\frac{\Gamma; \Delta, x : A \vdash t : B}{\Gamma; \Delta \vdash (\lambda x : A.t) : A \multimap B}\ (\multimap\text{-I}) \qquad \frac{\Gamma; \Delta_1 \vdash u : A \multimap B \quad \Gamma; \Delta_2 \vdash t : A}{\Gamma; \Delta_1, \Delta_2 \vdash (u\ t) : B}\ (\multimap\text{-E})$$

$$\frac{\Gamma; \varnothing \vdash t : A}{\Gamma; \varnothing \vdash\ !t : !A}\ (!\text{-I}) \qquad \frac{\Gamma; \Delta_1 \vdash u : !A \quad \Gamma, x : A; \Delta_2 \vdash t : B}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let } !x : A \textbf{ be } u \textbf{ in } t : B}\ (!\text{-E})$$

Figure 2.1: Typing Rules for DILL linear lambda calculus

There are two rules for typing variables, corresponding to intuitionistic (Int-Var) and linear variables (Lin-Var) respectively. In both cases we allow extra intuitionistic variables to be present in the environment, as we are free to ignore them. Conversely, linear variables must be used exactly once, so there can't be any present when typing a non-linear variable, and there can't be any extra ones present when typing a linear variable. Hence the linear contexts for the two rules are $\varnothing$ and $x : A$ respectively.

The unit type, $\mathbb{I}$, is inhabited by a single term $*$. The term $\textbf{let } * \textbf{ be } t \textbf{ in } u$ allows for the *elimination* of any term t of type $\mathbb{I}$, as shown by the rule (Unit-E). Every

type constructor has an introduction rule and an elimination rule, suffixed **-I** and **-E** respectively.

The product type $A \otimes B$ represents a pair of values and is inhabited by terms of the form $t \otimes u$ if $t : A$ and $u : B$, as demonstrated by its introduction rule ($\otimes$-I). In order to guarantee that linear variables are used exactly once in $t \otimes u$, it is required that the linear context (Δ_1, Δ_2) is the result of joining the two variable-disjoint linear contexts of t and u. Alternately, we can view this as the environment for $t \otimes u$ being *split* into the two environments for t and u. This is the **context splitting** operation that is at the core of many substructural type systems, and is the focus of our Coq library. Note that we use both the words *environment* and *context* to refer to typing environments.

The elimination rule for products ($\otimes$-E) also makes use of context splitting to ensure linearity. The rule states that if we have a term $u : A \otimes B$ and a term $t : C$ that is well-typed with free variables $x : A$ and $y : B$, then the expression that makes the components of u available as x and y in t also has type C. DILL uses the notation **let** $x \otimes y : A \otimes B$ **be** u **in** t for this expression, which is equivalent to `let (x, y) = u in t` in Haskell syntax. The intuitionistic context Γ from the **let** expression is available to both premises, whilst the linear context (Δ_1, Δ_2) is split so that some variables are used to type u (Δ_1), and others are used to type t in combination with the components of u $(\Delta_2, x : A, y : B)$.

The *lollipop* type, $A \multimap B$, is the type of functions that consume a linear value of type A and produce a value of type B. As we shall see shortly, regular functions $A \to B$ that don't destructively consume their input can be encoded as $!A \multimap B$. Introducing a new value of lollipop type is done by writing a λ-abstraction, $(\lambda x : A.\ t)$, which is well-typed only if its body t is well-typed under a linear environment extended with the type of the binder: $\Delta, x : A$. This rule, ($\multimap$-I) is identical to the rule for typing λ-abstractions in the Simply-Typed Lambda Calculus (STLC) except for the fact that it places the binder in a linear context.

The elimination rule for lollipop types ($\multimap$-E) uses a function application $(u\ t)$, where $u : A \multimap B$ and $t : A$. The context splitting is identical to the context splitting in the

($\otimes$-I) rule, with the linear context for $(u\ t)$ splitting into two sub-contexts for each of the sub-expressions u and t.

Finally we come to the *bang* type, $!A$, which allows the embedding of intuitionistic terms within the language. The introduction rule for bang types, (!-I), states that if a term t can be assigned the type A without reference to any linear variables, then we can construct a term $!t : !A$ which represents a duplicable version of t. Intuitively, this makes sense, as we can refer to the variables in the intuitionistic context for t as many times as we like whilst creating copies via $!t$.

To understand how terms of bang type become usable as duplicates requires consideration of the associated elimination rule, (!-E). This rule allows a term $u : !A$ to be destructured so that its "inner" term becomes available in the *non-linear* context as re-usable assumption $x : A$. If $u : !A$ was determined using the introduction rule for bang types such that $u = !v$ for some v, then the let binding **let** $!x : A$ **be** u **in** t binds a new name x to v and makes it available in the intuitionistic context for t.

Alternatively, it's possible for a term u to have type $!A$ without an application of the bang introduction rule. This occurs, for example, when writing a function to duplicate its input as a pair. Terms of linear type can't be duplicated, so the argument to this function must have type $!A$ for some type A. We must then use a bang let-binding, and (!-E) to bring a duplicable version of the binder variable into the intuitionistic context. The term we would like to type is therefore: $(\lambda x : !A.\ \text{let } !y : A \text{ be } x \text{ in } (!y \otimes !y))$. The typing derivation is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing; x : !A \vdash x : !A}\ \text{(Lin-Var)} \quad \dfrac{\dfrac{\dfrac{}{y : A; \varnothing \vdash y : A}\ \text{(Int-Var)}}{y : A; \varnothing \vdash !y : !A}\ \text{(!-I)} \quad \dfrac{\dfrac{}{y : A; \varnothing \vdash y : A}\ \text{(Int-Var)}}{y : A; \varnothing \vdash !y : !A}\ \text{(!-I)}}{y : A; \varnothing \vdash (!y \otimes !y) : (!A \otimes !A)}\ \text{($\otimes$-I)}}{\varnothing; x : !A \vdash \text{let } !y : A \text{ be } x \text{ in } (!y \otimes !y) : (!A \otimes !A)}\ \text{(!-E)}}{\varnothing; \varnothing \vdash (\lambda x : !A.\ \text{let } !y : A \text{ be } x \text{ in } (!y \otimes !y)) : !A \multimap (!A \otimes !A)}\ \text{($\multimap$-I)}$$

Note how we rely on (Lin-Var), rather than the bang introduction rule (!-I), to form the judgement $\varnothing; x : !A \vdash x : !A$. We also see the bang elimination rule in action here,

adding the new name y for x into the intuitionistic context for $!y \otimes !y$.

In the Simply-Typed Lambda Calculus, an equivalent duplication function would have type $A \to (A \otimes A)$. In fact, it is possible to embed all intuitionistic (STLC) terms and types in linear logic using reasoning similar to the above [Bar96]. Intuitionistic types A can be translated to duplicable bang types $!A$, and functions $A \to B$ can be translated to $!A \multimap B$.

2.4.1 Structural Rules

Philip Wadler gives a similar typing derivation for a duplication function in his 1993 presentation of linear logic [Wad93], although his system differs from DILL in a few key ways.

In Wadler's system, a single typing environment is used, containing both linear assumptions of the form $\langle x : A \rangle$ and intuitionistic ones of the form $[x : A]$. These are conceptually equivalent to assumptions in the respective linear and intuitionistic contexts of DILL. For rules that require the context to consist entirely of intuitionistic assumptions, Wadler uses the notation $[\Gamma]$, which is approximately equivalent to a $\Gamma; \varnothing$ pair of contexts in DILL. When formalising linear logic with an interactive theorem prover such as Coq, the dual-context approach taken by DILL is preferable to Wadler's approach because of the clear and forced separation of the two worlds – intuitionistic and linear. For example, Wadler's system would require a Coq data-type to differentiate the two types of assumptions, and an additional predicate to state if a context contains only intuitionistic assumptions. A further difference between the two systems is that Wadler's renders the structural rules that lend their name to *substructural* type systems explicit.

To save writing the entire set of typing rules, which are very similar to the ones for DILL already presented and the upcoming rules for L^3, we cherry-pick a few of the structural rules to demonstrate our point, and rely on the approximate translation described above and the original paper [Wad93] to provide the full details.

Unlike DILL, both kinds of variables in Wadler's LLC require their typing contexts to be empty except for the variable of interest:

$$\frac{}{[x : A] \vdash x : A}\ \text{(Int-Var')} \qquad \frac{}{\langle x : A \rangle \vdash x : A}\ \text{(Lin-Var')}$$

Duplication and discarding of intuitionistic variables is then enabled by two *structural* rules, Contraction and Weakening:

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

Contraction makes use of substitutions ($u[x/y, x/z]$) to form terms with two occurrences of the intuitionistic variable x. Weakening allows a non-vital variable and type to be introduced from nowhere, or, when reasoning backwards, it allows an unused variable to be discarded in order to type a sub-term.

In DILL the contraction and weakening rules for intuitionistic variables are implicit in how the intuitionistic context is managed – there can be additional intuitionistic assumptions present when typing variables (weakening), and intuitionistic assumptions can be duplicated freely (contraction). In contrast, in Wadler's system all duplication and discarding happens via (Contraction) and (Weakening). With the exception of the structural rules, Wadler's typing contexts behave entirely linearly, and are **split** in the same way linear contexts in DILL are split.

If the use of structural rules is restricted, we get a substructural type system. Without contraction, *variables can only appear once in a term.* Without weakening, *all variables in the context must be used in the term.* Banning both contraction and weakening results in a **linear** type system where variables must be used exactly once. The systems we've seen so far, DILL and Wadler's LLC, are both linear in that contraction and weakening are banned for the non-intuitionistic terms of the language. Under the

Curry-Howard correspondence, linear type systems correspond to linear logic, which was first introduced by Girard [Gir87].

If contraction is restricted but weakening is allowed, the result is an **affine** type system, corresponding to an affine logic (Grishin, 1974, Russian; recently [TP11]). Variables in affine type systems can be thrown away but not duplicated, so every variable is used *at most once*.

2.4.2 Summary of the Linear Lambda Calculus

In this section we've seen how typing rules can be used to enforce constraints on the use of variables, by examining Dual Intuitionistic Linear Logic. We have seen that intuitionistic terms can be embedded in substructural languages via bang types and the careful management of typing contexts. The importance of **context splitting** to divide linear assumptions between sub-terms has also been highlighted. In the next sections we examine more complex calculi that build on the linear lambda calculus to model useful programming constructs like mutation and memory allocation.

2.5 Operational Semantics and Type Soundness

Up until this point, we have described only the syntax and static semantics (typing rules) of linearly-typed programming languages, with only vague notions of how terms behave dynamically at run-time. To now describe the dynamic behaviour of languages we turn to *structural operational semantics*, a mathematical formalism for the step-wise evaluation of terms, also known as small-step semantics.

To define the small-step semantics of a language, we inductively define a relation on pairs of terms, denoted $t \Rightarrow t'$. The relation needn't be strictly on pairs of terms, and many small-step semantics also thread through a global state or *store*, σ, so that the stepping relation ends up being: $(\sigma, t) \Rightarrow (\sigma', t')$. In small-step semantics each step $t \Rightarrow t'$ is intended to represent a single step of the computation, in contrast to big-step

natural semantics where the values that terms evaluate to are stated directly, as in $t \Downarrow v$ [Gun92].

The β-rule of the λ-calculus can be stated for DILL as:

$$\frac{}{(\lambda x : A.\ t)\ v \Rightarrow t[v/x]}\ (\beta)$$

Here we use the variable v to indicate a *value* of the language. Values are terms that are fully evaluated according to the small-step semantics [Pie02] – formally, they are terms that are in normal form with respect to the stepping relation ($\Rightarrow$). By specifying that β-reduction can only occur if the argument is a value we have fixed the reduction strategy to call-by-value. Other reduction strategies can be similarly encoded, but we consider only call-by-value here, as it's simple and well-suited to low-level programming, as evidenced by the literate surveyed.

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. Stated formally, the soundness lemma is:

$$\frac{\varnothing \vdash e : \tau \quad e \Rightarrow^* e'}{(\exists e''.\ e' \Rightarrow e'') \vee e' \text{ is a value}}\ \text{(Type Soundness)}$$

The notation ($\Rightarrow^*$) represents zero or more applications of ($\Rightarrow$). Our definition of stuckness states that a term e' is not stuck if either it can step to some other term e'', or it is a value of the language.

Soundness can be proved via two supporting lemmas called *progress* and *preservation*, using an approach introduced by Wright and Felleisen [WF94].

The progress property holds if all well-typed closed terms are either values, or can take a step.

$$\frac{\varnothing \vdash e : \tau}{(\exists e'.\ e \Rightarrow e') \vee e \text{ is a value}}\ \text{(Progress)}$$

The preservation, or *subject reduction*, property holds if well-typed terms retain their type during evaluation.

$$\frac{\varnothing \vdash e : \tau \quad e \Rightarrow e'}{\varnothing \vdash e' : \tau} \text{ (Preservation)}$$

Our proof-of-concept Coq formalisation of a linear lambda calculus makes use of progress and preservation lemmas to establish type soundness. There are other formalisms and techniques that can be used to demonstrate soundness and similar properties, but we restrict our attention here to syntactic proofs by progress and preservation.

2.6 The Linear Language with Locations, L^3

We now turn our attention to *The Linear Language with Locations*, L^3 [MAF05], which extends the linear lambda calculus with strong destructive updates and explicit memory management. In a language with references or pointers, a destructive update is a re-assignment of a value pointed to by a pointer. A *strong* destructive update is one that may also change the *type* of the value pointed to by the pointer. Destructive updates are a common feature of imperative languages, and their addition to the linear lambda calculus brings us a step closer to a sound and usable programming language for low-level programming. L^3 itself is not such a language, but could serve as a foundation for a language with more features (e.g. polymorphism). Its designers created it as a foundational calculus for strong updates, which can be used to model type-varying CPU registers.

As for Dual Intuitionistic Linear Logic, we begin with a description of the syntax of the language, in Figure 2.2.

Many of the terms and types have the same meaning as in DILL. Functions, variables, products and the unit value are all the same. L^3's complete specification also includes small-step operational semantics, for which the set of values denoted by meta-variable v are normal forms.

$$
\begin{array}{rcl}
x, y & \in & \textbf{Vars} \\
l & \in & \textbf{LocConsts} \\
\rho & \in & \textbf{LocVars} \\
\eta & ::= & l \mid \rho \\
A & ::= & \mathbb{I} \mid A \otimes B \mid A \multimap B \mid !A \mid \textbf{Ptr}\ \eta \mid \textbf{Cap}\ \eta\ A \mid \forall \rho.\ A \mid \exists \rho.\ A \\
t, u & ::= & * \mid \textbf{let}\ * \ \textbf{be}\ t\ \textbf{in}\ u \mid \\
 & & t \otimes u \mid \textbf{let}\ x \otimes y\ \textbf{be}\ t\ \textbf{in}\ u \mid \\
 & & x \mid \lambda x.\ t \mid (t\ u) \mid \\
 & & !v \mid \textbf{let}\ !x\ \textbf{be}\ t\ \textbf{in}\ u \mid \textbf{dupl}\ t \mid \textbf{drop}\ t \mid \\
 & & \textbf{ptr}\ l \mid \textbf{cap}\ l \mid \textbf{new}\ t \mid \textbf{free}\ t \mid \textbf{swap}\ t\ u \\
 & & \Lambda \rho.\ t \mid t[\eta] \mid \ulcorner \eta, t \urcorner \mid \textbf{let}\ \ulcorner \rho, x \urcorner\ \textbf{be}\ t\ \textbf{in}\ u \\
v & ::= & * \mid v_1 \otimes v_2 \mid x \mid \lambda x.\ t \mid !v \mid \textbf{ptr}\ l \mid \textbf{cap}\ l \mid \Lambda \rho.\ t \mid \ulcorner \eta, v \urcorner
\end{array}
$$

Figure 2.2: Syntax for The Linear Language with Locations

The main addition is the set of primitives for allocating and managing memory: **new** t, **free** t and **swap** t u. Each piece of memory allocated is at a constant location l, drawn from a set of location constants **LocConsts** which can be considered memory addresses. The precise locations are hidden from the programmer by existential types $\exists \rho.\ A$, for location variables $\rho \in$ **LocVars**.

L^3 is typical of other modern calculi in that it separates resources like pointers from *capabilities* to use resources. A pointer value **ptr** l : **Ptr** l is unusable without a capability to read from and write to it: **cap** l : **Cap** l A. Capabilities are linear values, whilst pointers are duplicable. To see how this is enforced, and the destructive updates that it enables, we now considering the typing rules for L^3, as in Figures 2.3 and 2.4.

$$\frac{}{\Delta; \varnothing \vdash * : \mathbb{I}} \text{ (Unit-I)}$$

$$\frac{\Delta; \Gamma_1 \vdash t : \mathbb{I} \quad \Delta; \Gamma_2 \vdash u : A}{\Delta; \Gamma_1, \Gamma_2 \vdash \textbf{let } * \textbf{ be } t \textbf{ in } u : A} \text{ (Unit-E)}$$

$$\frac{FLV(A) \subseteq \Delta}{\Delta; x : A \vdash x : A} \text{ (Var)}$$

$$\frac{\Delta; \Gamma_1 \vdash t : A \quad \Delta; \Gamma_2 \vdash u : B}{\Delta; \Gamma_1, \Gamma_2 \vdash t \otimes u : A \otimes B} \text{ ($\otimes$-I)}$$

$$\frac{\Delta; \Gamma_1 \vdash t : A \otimes B \quad \Delta; \Gamma_2, x_1 : A, x_2 : B \vdash u : C}{\Delta; \Gamma_1, \Gamma_2 \vdash \textbf{let } x_1 \otimes x_2 \textbf{ be } t \textbf{ in } u : C} \text{ ($\otimes$-E)}$$

$$\frac{\Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.\ t : A \multimap B} \text{ ($\multimap$-I)}$$

$$\frac{\Delta; \Gamma_1 \vdash t : A \multimap B \quad \Delta; \Gamma_2 \vdash u : A}{\Delta; \Gamma_1, \Gamma_2 \vdash (t\ u) : B} \text{ ($\multimap$-E)}$$

$$\frac{\Delta; !\Gamma \vdash v : A}{\Delta; !\Gamma \vdash !v : !A} \text{ (!-I)}$$

$$\frac{\Delta; \Gamma_1 \vdash t : !A \quad \Delta; \Gamma_2, x : A \vdash u : B}{\Delta; \Gamma_1, \Gamma_2 \vdash \textbf{let } !x \textbf{ be } t \textbf{ in } u : B} \text{ (!-E)}$$

$$\frac{\Delta; \Gamma \vdash t : !A}{\Delta; \Gamma \vdash \textbf{dupl } t : !A \otimes !A} \text{ (Dupl)}$$

$$\frac{\Delta; \Gamma \vdash t : !A}{\Delta; \Gamma \vdash \textbf{drop } t : \mathbb{I}} \text{ (Drop)}$$

Figure 2.3: Typing Rules for The Linear Language with Locations (Part I)

$$\frac{\Delta; \Gamma \vdash t : A}{\Delta; \Gamma \vdash \mathbf{new}\ t : \exists \rho.\ \mathbf{Cap}\ \rho\ A \otimes\ !(\mathbf{Ptr}\ \rho)}\ (\text{New})$$

$$\frac{\Delta; \Gamma \vdash t : \exists \rho.\ \mathbf{Cap}\ \rho\ A \otimes\ !(\mathbf{Ptr}\ \rho)}{\Delta; \Gamma \vdash \mathbf{free}\ t : \exists \rho.\ A}\ (\text{Free})$$

$$\frac{\Delta; \Gamma_1 \vdash t : \mathbf{Ptr}\ \rho \quad \Delta; \Gamma_2 \vdash u : (\mathbf{Cap}\ \rho\ A) \otimes B}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{swap}\ t\ u :\ (\mathbf{Cap}\ \rho\ B) \otimes A}\ (\text{Swap})$$

$$\frac{\Delta, \rho; \Gamma \vdash t : A}{\Delta; \Gamma \vdash \Lambda \rho.\ t : \forall \rho.\ A}\ (\text{LFun})$$

$$\frac{\Delta; \Gamma \vdash t : \forall \rho.\ A \quad \rho' \in \Delta}{\Delta; \Gamma \vdash t[\rho'] : A[\rho'/\rho]}\ (\text{LApp})$$

$$\frac{\rho' \in \Delta \quad \Delta; \Gamma \vdash t : A[\rho'/\rho]}{\Delta; \Gamma \vdash \ulcorner \rho', t \urcorner : \exists \rho.\ A}\ (\text{LPack})$$

$$\frac{\Delta; \Gamma_1 \vdash t : \exists \rho.\ A \quad FLV(B) \subseteq \Delta \quad \Delta, \rho; \Gamma_2, x : A \vdash u : B}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let}\ \ulcorner \rho, x \urcorner\ \mathbf{be}\ t\ \mathbf{in}\ u : B}\ (\text{Let-LPack})$$

Figure 2.4: Typing Rules for The Linear Language with Locations (Part II)

L^3's typing judgements include two contexts, one for locations (Δ) and another for types (Γ), as in $\Delta; \Gamma \vdash t : A$. All variables and their types are stored in one typing context, which makes L^3 more similar to Wadler's LLC than DILL, despite the syntactic similarity. The location context Δ is a sequence of location variables that are currently in scope, like variables in a typing context without the type information: $\Delta ::= \varnothing \mid \Delta, \rho$.

As in DILL, **context splitting** is used extensively to ensure that each variable appears exactly once in a term. The typing rules for unit types, product types ($\otimes$) and functions ($\multimap$) are almost identical to the rules for their counterparts in Wadler's LLC, and DILL, modulo the different number of contexts.

The rule for variables, (Var), is similar to Wadler's, except that the *free location variables* of the variable's type, $FLV(A)$, must be present in the location context. Another

interesting difference is that L^3 has only one rule for variables, rather than two like DILL and Wadler's LLC. The reason for this is that L^3 doesn't differentiate between linear and intuitionistic assumptions. Wadler motivates the two types of assumptions by giving an example of how the proof reduction rule equivalent to β-reduction becomes unsound if contraction and weakening are allowed for assumptions of the form $\langle x : !A \rangle$ (or just $x : !A$ in L^3 syntax). This doesn't pose a problem to L^3 because there are no explicit contraction and weakening rules – the same functionality is instead provided by **dupl** t and **drop** t primitives. As a result, a variable is considered intuitionistic (and duplicable) if it has type $!A$ for some A.

The **drop** and **dupl** primitives form part of L^3's handling of intuitionistic values using the bang type. The introduction rule for bang types is almost the same as in DILL, except that only values v are permitted, and an intuitionistic context $!\Gamma$ is now just one where all assumptions are of the form $x : !A$. The elimination rule is also similar, except that the context which the newly bound variable is introduced into is linear, which means the variable can only be used once after the rule is applied. The duplication primitive mitigates this restriction by explicitly transforming a value of type $!A$ into a pair of values with the same type, $!A \otimes !A$. The components of the duplicate pair can then be given names and introduced into the typing context by the elimination rule for products, ($\otimes$-E). This duplicating of intuitionistic values is equivalent to a contraction typing rule.

Further, there is an equivalent to a weakening rule for intuitionistic terms, in the form of the **drop** primitive, which allows a term to be discarded. Given a term t with type $!A$, dropping it results in a term **drop** t with type unit $\mathbb{I}$.

The remaining typing rules relate to L^3's primitives for memory allocation. By the (New) rule, we see that a term t can be assigned a reference and accompanying capability by the **new** t term. The type of **new** t is an existential type $\exists \rho.\ (\mathbf{Cap}\ \rho\ A \otimes !(\mathbf{Ptr}\ \rho))$, which hides the precise location from the programmer. As can be seen from the (Let-LPack) rule, this existential package can be unpacked by a **let** $\ulcorner \rho, x \urcorner$ **be** t **in** u term, causing the location ρ to become available in the location context, and a variable

for the pointer and capability to become available in the typing context.

Note that the capabilities produced by the **new** construct have linear type $\mathbf{Cap}\ \rho\ A$, while the pointers themselves are non-linear and duplicable $!(\mathbf{Ptr}\ l)$. Further, the type of the pointer doesn't mention the type of the value it points to. These two properties enable strong destructive updates, as the unique capability can be passed to the **swap** construct, which replaces the value at a location and evaluates to a pair containing the old value and a capability with the new value's type. The (Swap) rule demonstrates this.

Finally, the **free** primitive, which is responsible for de-allocating a piece of memory allocated with **new**. A term **free** t is well-typed if the term t is a capability-and-pointer package as produced by **new**. The value of **free** t after de-allocation is an existential package containing the value that was stored at the location, with type $\exists\rho.\ A$.

The run-time interpretations of the above operations are provided by a small-step operational semantics that makes use of a store σ mapping locations l to closed values v. We display only a selection of the reduction rules, which are lifted to small-step operational semantics by evaluation contexts. For the full set of reduction rules and evaluation contexts, see the technical report accompanying L^3 [MAF04].

$$\text{(new)} \quad (\sigma, \mathbf{new}\ v) \Rightarrow (\sigma \uplus \{l \mapsto v\}, \ulcorner l, \mathbf{cap}\ l \otimes !(\mathbf{ptr}\ l)\urcorner)$$

$$\text{(free)} \quad (\sigma \uplus \{l \mapsto v\}, \mathbf{free}\ \ulcorner l, \mathbf{cap}\ l \otimes !(\mathbf{ptr}\ l)\urcorner) \Rightarrow (\sigma, \ulcorner l, v\urcorner)$$

$$\text{(swap)} \quad (\sigma \uplus \{l \mapsto v_1\}, \mathbf{swap}\ (\mathbf{ptr}\ l)\ (\mathbf{cap}\ l \otimes v_2)) \Rightarrow (\sigma \uplus \{l \mapsto v_2\}, \mathbf{cap}\ l \otimes v_1)$$

$$\text{(let-lpack)} \quad (\sigma, \mathbf{let}\ \ulcorner \rho, x\urcorner\ \mathbf{be}\ \ulcorner l, v\urcorner\ \mathbf{in}\ t) \Rightarrow (\sigma, t[l/\rho][v/x])$$

These rules codify the intuition for the constructs given above. For example, (new) introduces a new location l into the store, mapped to the appropriate value, whilst (free) performs the inverse. Capabilities are threaded throughout to ensure that exclusive rights to update the memory location are held.

For their proof of soundness, the L^3 authors don't use a syntactic proof. Instead, they

provide a *semantic interpretation* of the types and prove that well-typed expressions correspond to true logical statements about the interpretations of the types. A consequence of this approach is that the typing rules don't contain rules for intermediate values like **cap** l, which makes a syntactic proof more complicated, as discussed in §4.2.

2.7 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 highlighted by the contrast between *dereliction* and the rule that we'll refer to as *uniqueness removal* present in Clean's type system. In a linearly typed language, dereliction refers to the ability to convert intuitionistic values into linear ones, for example by constructing a function with type $(!A \multimap A)$. In DILL, the following dereliction function is well-typed:

$$\varnothing; \varnothing \vdash (\lambda x : !A.\ \textbf{let}\ !y : A\ \textbf{be}\ x\ \textbf{in}\ y) : !A \multimap A$$

Now, imagine that we treat linear terms as unique, and intuitionistic ones as non-unique. As noted by Edsko de Vries [dV08], the dereliction function above is **not sound** under this proposed equivalence. The "unique" value resulting from dereliction is not necessarily unique because other shared references (intuitionistic values of type $!A$) may still exist.

Further, the uniqueness removal rule in Clean allows unique values to be transformed into non-unique ones. In a linear context, this is analogous to writing a function with type $A \multimap !A$, which violates the guarantee that linear variables are only used once, and is impossible in DILL. 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. 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.

Despite uniqueness being distinct from linearity, Edsko's formalism also makes use of a context splitting operation to enforce constraints on the usage of variables. It differs slightly from the standard approach in that non-unique variables can be split onto both sides. A discussion about expressing de Vries' context splitting in terms of the standard approach is given in §4.1.

Finally, de Vries' work includes the only known mechanical proof of type soundness for a type system similar to Clean's. The proof is syntactic, and is encoded in Coq. It makes use of the *locally nameless* approach to variable naming that is discussed in

§2.11.2.

2.8 Systems of Capabilities

2.8.1 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 proof of correctness for this work exists, 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.

2.8.2 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.

For context splitting, SSPHS makes use of a *multiplicity environment* which records the number of available copies of each variable. Each variable is marked as having 0, 1 or ∞ copies available, encoding unavailable linear variables, available linear variables and intuitionistic variables respectively. Pottier exploits the fact that multiplicity environments form a *separation algebra*, and unifies their treatment with the treatment of general resources (including regions). In this system, context splitting is the division of a context such that the number of copies of each variable is preserved.

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 §2.11.2). 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.

2.8.3 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.

2.9 Linear Dependent Types

Conor McBride's recent work on combining linear and dependent typing [McB16] is strikingly similar to Pottier's SSPHS in its treatment of typing contexts. In Conor's system, variables in the context are annotated by the number of occurrences available

at run-time. Linear variables are annotated with a **1** if available, or a **0** if they have already been used in a neighbouring part of the term. This allows types to depend upon linear values, by referring to the **0** available copies if necessary – a process Conor calls *contemplation*. As in Pottier's work, an infinity ∞ annotation is used for intuitionistic variables, and context splitting is the resource-preserving division of a context into two pieces.

2.10 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.

2.11 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 $e[v/x]$) to describe the semantics of function application:

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

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, it doesn't help our overall goal of creating a *general* framework for substructural languages, in which substitution of open terms should be possible.

To attain capture-avoiding substitution 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.

2.11.1 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.

2.11.2 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 $(\lambda.\ \hat{0})$, where a natural number annotated with a hat represents a de Bruijn index.

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 $\{x \mapsto \hat{2}, y \mapsto \hat{1}, z \mapsto \hat{0}\}$, the term $(\lambda x.\ (x\ y)\ z)$ would be encoded as $(\lambda.\ (\hat{0}\ \hat{2})\ \hat{1})$.

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 *lifting* operation. Lifting 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 lifting 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 lifting defined, the definition of substitution is straight-forward – we simply lift the free variables of the substituted term by 1 each time we move under a lambda.

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

These equations for lifting 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 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. Our library is an extension of DbLib, and our proof-of-concept makes successful use of it for substitution, as discussed in §3.4.

The alternative to DbLib would have been to use Arthur Charguéraud's *Engineering Formal Metatheory* (EFMT) library [ACP+08] for binding using a *locally nameless* 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, EFMT depends on Charguéraud's TLC library for *non-constructive* logic within Coq [Cha16]. We elected not to use this library, in order to keep the set of axioms minimal and to allow us to explore constructive logic.

2.12 Summary of Mechanisation Techniques

The following table (Figure 2.5) 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. 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]	✓	✓	✓	×	-	×	-
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.

Figure 2.5: Summary of Mechanisation Techniques

2.13 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). **Context splitting** plays a key role in almost all systems surveyed. 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. Several capability systems mention Alias Types [SWM00] and separation logic [Rey02] as foundational concepts, but we defer in-depth discussion of these to future work.

2.14 Evaluation Framework

In this section we describe some criteria for assessing the quality of work completed as part of this thesis. Given our goal of creating a general Coq library for the verification

of linearly-typed languages, what properties should our library ideally possess? We break the criteria into two subsections: **Conceptual Goals**, relating to the theoretical content of the library, and its generality and applicability to other languages; and **Implementation Goals** regarding the quality of the Coq library itself. Dividing the criteria in this way provides broad coverage of the quality of the work whilst allowing us to separate concerns.

2.14.1 Conceptual Goals

Our main conceptual goal is that the library be applicable and useful in the mechanical formalisation of numerous languages with substructural typing. We say *numerous*, rather than *all*, because new type systems with substructural influences are still being developed [McB16]. The library should be:

1. Applicable to the formalisation of L^3 and related systems.
2. Usable without modifications to the core definitions.
3. Usable with the addition of a minimal number of lemmas about the library's content.

Point (1) is a restatement of our main overarching goal specialised to L^3, which we argue is representative of a class of similar languages.

Point (2) expresses the ideal that the verifier of a new linearly-typed language should be able to use the library without making bespoke modifications to the core definitions and lemmas. Such modifications would indicate a lack of generality in the library's content. *Usable* here also means that the potential user of the library should not have to work around the library's inadequacies in convoluted ways.

Point (3) relates to the coverage provided by the library's lemmas. Ideally, all of the interesting relationships between its parts should be chronicled as lemmas within the library. We allow some flexibility, to acknowledgement that the ideal is typically

very time-consuming to achieve. We believe it is reasonable for users of the library to discover a small number of missing relationships between the library's components which they can then contribute *upstream* for general use.

2.14.2 Implementation Goals

The construction of a large Coq proof, as in the development of any complex piece of software, demands attention to quality of design and implementation. Due to the lack of well established engineering techniques for interactive proofs we describe and justify some of our own criteria here. Discussion of Adam Chlipala's automated style of theorem proving is given as part of a larger discussion about proof automation in the Evaluation chapter (§4.4.3).

Firstly, our Coq proofs should be **easy to read and understand**. By this we mean that additional complexity or obfuscation that detracts from the *intent* of the proof should be kept to a minimum. Coq proofs differ from most other bits of code in that they are often difficult to read without knowing the goal and hypotheses at each step, so our criteria for readability must take this into account.

Some heuristics we can use to assess **readability** are:

- Different levels of indentation for goals and sub-goals.
- Minimal nesting of cases and sub-cases; prefer lemmas.
- Meaningful variable names at every step; avoid auto-generated names.
- Short proofs of lemmas.

These heuristics are adapted from common software engineering practice, and should be familiar to anyone with a programming background. Short proofs of lemmas are like short functions, and preferring a small lemma to more code in the same lemma is like preferring a utility function to more code in the same function body. The meaningful

naming of variables *at every step* links back to our altered definition of readability where we imagine that the reader is stepping through the proof examining the goals and hypotheses at each step.

One aim of readable proofs is to convey key insights, but readability also aids **maintainability**. Proofs, particularly libraries of proofs, are not static entities and must be constructed so that they can be updated as easily as possible as features are added and new versions of related software are released. Between different versions of the theorem prover the behaviour of tactics can change in ways that are not backwards compatible. References to automatically named variables are considered fragile as a change in the automatic name generation can invalidate the reference and all subsequent proof steps. Some additional criteria to aid **maintainability** are:

- No repetition of proof script.
- Markers to enforce different cases and sub-cases.

Avoiding copied sections of proof script aids maintainability in an obvious way – changing one section of code doesn't require changing all of its copies.

Enforced case markers improve the clarity of error messages and the general debugging experience. By *enforced* we mean that the case markers only allow a new case to begin if the existing case has already been solved. Without case markers a tactic failure early in a proof script can cause tactics on subsequent lines to be applied to the wrong goals, leading to confusing error messages. With case markers, the cases that fail are isolated from their surroundings.

An assessment of the work according to this evaluation framework is given in the Evaluation chapter (chapter 4). We turn our attention now to a detailed description of the work.

Chapter 3

Own Work

3.1 Research Questions

The primary research question that this thesis attempts to answer relates to the efficient development of mechanically verified proofs about languages with substructural typing. Specifically,

- What are the common properties of all proofs about substructural languages, and can these properties be exploited to avoid duplicated work in the context of interactive theorem proving?

In the context of Coq, this is can concretely be phrased:

- Is it possible to write Coq lemmas and definitions which are useful for the verification of numerous substructural type systems?

3.2 Outline of Own Work

As part of this thesis, the following work has been completed:

- Development of general definitions and lemmas about context splitting and other actions on typing environments.
- Proof of type soundness for a linear lambda calculus.

The proof of type soundness for the linear lambda calculus consists of a collection of lemmas and definitions, culminating in proof of **progress** and **preservation** which together establish soundness.

We begin by defining the language for which we have established type soundness, and continue with an in-depth discussion of how it was formalised in Coq. The definitions and lemmas provided by the library are motivated by the proof and are discussed as they arise.

3.3 Purely Linear Lambda Calculus

The language formalised is a wholly linear subset of DILL, extended with uninterpreted primitive types. For brevity, call this language PLLC, for the Purely Linear Lambda Calculus. PLLC provides just enough features to reason about linearity in interesting ways, while not being too labour-intensive to formalise. Its syntax is:

$$\begin{array}{rcl} x & \in & \textbf{Vars} \\ s & \in & \textbf{Strings} \\ A, B & ::= & \mathbb{I} \mid A \multimap B \mid \textbf{TyPrim}\ s \\ t, u & ::= & * \mid x \mid (\lambda x : A.\ t) \mid (t\ u) \mid \textbf{TPrim}\ s \\ v & ::= & * \mid x \mid (\lambda x : A.\ t) \mid \textbf{TPrim}\ s \end{array}$$

The primitive types provide some types other than the unit type for the function type constructor to act on. Their inclusion has minimal impact on the proof of soundness and they can be removed without altering the outcome. All of the other terms have the same meaning as in DILL. The typing rules for PLLC are shown in Figure 3.1.

$$\frac{}{x : A \vdash x : A}\ (\text{Lin-Var}) \qquad \frac{}{\varnothing \vdash * : \mathbb{I}}\ (\text{Unit})$$

$$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x : A.\ t : A \multimap B}\ (\multimap\text{-I}) \qquad \frac{\Delta_1 \vdash u : A \multimap B \quad \Delta_2 \vdash t : A}{\Delta_1, \Delta_2 \vdash (u\ t) : B}\ (\multimap\text{-E})$$

$$\frac{}{\varnothing \vdash \mathbf{TPrim}\ s_1 : \mathbf{TyPrim}\ s_2}\ (\text{Prim})$$

Figure 3.1: Typing Rules for PLLC

With the exception of the rule for primitives, PLLC's typing rules are a subset of DILL's, with the intuitionistic contexts Γ removed. Hence, in PLLC, every variable is linear and every function consumes its input. In the ($\multimap$-E) rule, note the use of context splitting (Δ_1, Δ_2) to divide the variables available to an application. The rule for primitives allows any primitive term to be assigned any primitive type, the intention being that primitive types are handled by a separate mechanism, as in a compiler.

The operational semantics for PLLC are derived from call-by-value semantics for the Simply-Typed Lambda Calculus [PCG+15]. The three rules, shown in Figure 3.2 include the (β) rule for computation by substitution, and two rules for evaluating applications. The (StepApp1) rule states that if the first half of an application u can step to a term u', then the entire application can take a step by stepping u: $(u\ t) \Rightarrow (u'\ t)$. The (StepApp2) rule is a similar rule for the second sub-term of an application, and applies only once the first sub-term has been reduced to a value.

By enforcing the restriction that the first sub-term of the application in (StepApp2) is a value, an evaluator never needs to choose which side should be evaluated first, thus removing non-determinism. If we include the (β) rule in our consideration we see that the stepping relation as a whole is deterministic, although this property isn't verified in our Coq proofs.

As is somewhat standard in syntax-driven Coq proofs about programming languages [PCG+15], we use inductive definitions for the terms, types and typing judgements

$$\frac{}{((\lambda x : A.\ t)\ v) \Rightarrow t[v/x]}\ (\beta)$$

$$\frac{u \Rightarrow u'}{(u\ t) \Rightarrow (u'\ t)}\ \text{(StepApp1)}$$

$$\frac{t \Rightarrow t'}{(v\ t) \Rightarrow (v\ t')}\ \text{(StepApp2)}$$

Figure 3.2: Operational Semantics for PLLC

of the language. Translating the semi-formal description of the language into Coq definitions is straight-forward once a few key parts of the representation are decided upon. Specifically:

- How do we represent the names of variables? As discussed in §2.11, we elect to use François Pottier's DbLib library for de Bruijn indices.
- How is substitution implemented? DbLib provides a substitution function given a few basic properties of a language. The implementation of this is discussed in the next section, §3.4.
- How are typing contexts represented? How is context splitting implemented? These questions are the topic of the upcoming sections: §3.5, §3.6, §3.7.

Discussion of the exact representation of terms, typing judgements and operational semantics is given in the Appendix, sections §A.1, §A.2, §A.3.

Type soundness is the property that we would like to establish to show that PLLC is a well-behaved programming language. We do this by proving progress and preservation lemmas, as defined in §2.5. The statement of these lemmas in Coq is given in §3.8 and §3.9, as well as consideration of how they were proved.

The proofs of progress and preservation are *language-specific* in that they only apply to PLLC. We are primarily interested in the *universal* lemmas which can be used

independently of PLLC and may be applicable to the verification of other linearly typed languages. In general, any lemmas that reference the terms, semantics or typing rules of PLLC are language specific, while lemmas about context splitting and typing contexts are universal and are part of our library.

3.4 Integrating with DbLib

DbLib provides functions for substitution and lifting that abstract over the manipulation of de Bruijn indices. Client libraries wanting to make use of DbLib need only implement a few fundamental operations via Coq's *type-classes* [CS12]. For the linear lambda calculus we can use essentially the same definitions as for the simply typed lambda calculus, which are provided as an example with DbLib.

First, we must inform DbLib which of our term constructors is for variables. DbLib allows the types of *values* (`V`) and *terms* (`T`) to differ, but we don't make use of this capability, instead using `term`s everywhere and the `value` predicate. The type-class has the following definition in DbLib:

```
Class Var (V : Type) := {
  var: nat -> V
}.
```

Our instance is straight-forward:

```
Instance Var_term : Var term := {
  var := TVar
}.
```

To convey how variables are bound and scoped, we must implement DbLib's `Traverse` type-class, which has a single function called `traverse`. From the DbLib documentation:

> `traverse` can be thought of as a semantic substitution function. The idea is, `traverse f l t` traverses the term `t`, incrementing the index `l` whenever a binder is entered, and, at every variable `x`, it invokes `f l x`. This produces a value, which is grafted instead of `x`.

The only binders in our linear lambda calculus are lambda abstractions, so our implementation of traverse only has to increment `l` when recursing below a `TAbs` constructor. This is the same as for the simply typed lambda calculus. If more substitution functions were required, as would be the case in a language with polymorphism and type substitutions, distinct named instances of `Traverse` could be created for each use.

```
Fixpoint traverse_term (f : nat -> nat -> term) l t :=
  match t with
  | TVar x =>
      f l x
  | TAbs t e =>
      TAbs t (traverse_term f (1 + l) e)
  | TApp e1 e2 =>
      TApp (traverse_term f l e1) (traverse_term f l e2)
  | _ => t
  end.
```

To ensure that the client's implementation of traverse behaves sensibly and can be manipulated accordingly, DbLib requires the implementation of five further type-classes that establish semantic properties of traverse. Also provided are five tactics for proving these properties automatically, which we found to be sufficient for our simple use-case.

Given these type-class definitions, DbLib provides a substitution function that we can make use of in the operational semantics for our language. The type of the substitution function is `V -> nat -> T -> T`, which is specialised to `term -> nat -> term -> term`. The Coq implementation of the (β) rule makes use of it, as shown in §A.3.

3.5 Representing Typing Contexts

Before defining an inductive predicate for our typing relation, a representation for typing environments must be selected. In semi-formal proofs, typing judgements are written $H \vdash e : \tau$, and the environment H is assumed to permit various operations such as looking up the type of a variable x (denoted $H[x]$) and (re)assigning a type to a variable x, (denoted $H[x \mapsto \tau]$).

As noted in the Background chapter, control over the actions permitted on typing contexts is at the core of substructural typing. Together with our choice to use de Bruijn indices for variable naming, this narrows down our choice of representation. We have the following options to consider:

- **Functions**: Some Coq formalisations of languages with structural typing [PCG+15] make use of functions to encode partial maps from variables to types. Assigning a type involves wrapping the existing environment in another conditional statement, as in `insert x` τ `H := fun y => if x = y then Some` τ `else H y`. This approach is unsuitable for substructural type systems because the function is opaque and can't be disassembled into two functions which are equivalent when combined. Given an arbitrary function, it is impossible to know that it is always going to consist of a conditional of the form shown, and therefore it is also impossible to extract any of the information about x, τ or the original H.
- **Lists of Types**: We could consider using a list of types so that the type for variable $\hat{i}$ is at index i. This is preferable to using a function because we can inspect and destructure a list, and can also perform induction. However, splitting an environment becomes problematic because we need to keep the type for variable $\hat{i}$ at index i, even if some or all of the types at indices less than i should no longer be available because they were assigned to the other side of the split. Essentially, if we are to use a list, we need a filler value to occupy the evacuated positions. This leads to our next option:
- **Lists of Optional Types**: What if we rather than using a list of types, we use a

list of `option type`, so that types which are no longer available are represented by `None` entries? This fulfils all of our requirements: we can look-up types, alter them, add new entries and split an environment so that variables and their types are divided between the two new environments.

For these reasons, our formalisation makes use of a list of optional types, as provided by the `env` type from DbLib. However, the lemmas about `env` provided by DbLib proved to be insufficient for reasoning about substructural typing rules. For example, DbLib treats the empty list `[] (nil)` as the only empty environment, when it is often useful to treat any number of `None`s as an empty environment. Further, context splitting defined on `Env` is general enough to be of use in multiple DbLib-based formalisations, so why not re-use this effort? The next two sections describe these two aspects of our formalisation, and this thesis's contribution to a general framework for substructural languages.

3.6 Emptiness

We define the following predicate for environments which is compatible with any environment from DbLib. We say that an environment is *empty* if it contains no typing information. Hence, the empty list environment (`nil`) is empty, as is any number of `None` values.

```
Inductive is_empty {A} : env A -> Prop :=
  | is_empty_nil : is_empty nil
  | is_empty_cons E (EmptyTail : is_empty E) : is_empty (None :: E).
```

The necessity of a predicate for emptiness arises from several uses of *inductive loading* in proofs of lemmas related to substitution and type preservation. Using `nil` environments proved to be too limiting, and there are several cases in the proof where an induction generates an empty environment like `raw_insert x None nil`, and has to apply an inductive hypothesis about empty contexts to it.

In order to be useful in the verification of languages, lemmas about the properties of the `is_empty` predicate and its interaction with other parts of the system are required. Of these lemmas, the ones involving interaction with DbLib were slightly more difficult to prove than those about simple Coq constructs. Examples of lemmas about emptiness are shown below, the full set can be found in the file `Linear/Empty.v` in the accompanying source code (§A.4).

```
Lemma empty_repeat : forall A (E : env A),
  is_empty E ->
  E = repeat (length E) None.
(* Proof by induction on is_empty E *)

Lemma empty_lookup : forall A x (E : env A),
  is_empty E ->
  lookup x E = None.
(* Proof by induction on x *)
```

3.7 Context Splitting

Context splitting is the operation by which a typing environment, implemented as a `list (option T)`, is split so that it may be used to type two expressions. We use the notation $E = E_1 \circ E_2$ to represent the splitting of E into two environments E_1 and E_2 such that each variable from E appears in only one of E_1 and E_2.

In PLLC, the typing rule for application requires that the context used to type $(e_1 e_1)$ can be split into two contexts that type e_1 and e_2 individually. This splitting causes variables from the typing context to be used at most once in the expression $(e_1 e_2)$, as each element of the context must be assigned to either the left or right sub-expression.

As we are working with a list, it makes sense to preserve the length of the context when splitting. As a base case we have $[] = [] \circ []$. If the list contains one or more elements, then that element can either be assigned to the left or to the right. In Coq, we define

an inductive predicate over contexts so that `context_split E E1 E1` $\equiv E = E_1 \circ E_2$.

```
Inductive split_single {A} : option A -> option A -> option A -> Prop :=
  | split_none : split_single None None None
  | split_left (v : A) : split_single (Some v) (Some v) None
  | split_right (v : A) : split_single (Some v) None (Some v).

Inductive context_split {A} : env A -> env A -> env A -> Prop :=
  | split_nil : context_split nil nil nil
  | split_cons E E1 E2 v v1 v2
      (SplitElem : split_single v v1 v2)
      (SplitPre : context_split E E1 E2) :
      context_split (v :: E) (v1 :: E1) (v2 :: E2).
```

3.7.1 Single Element Splits

One may wonder if the presence of an accompanying `split_single` predicate is necessary, as the same meaning can be achieved with the following definition that places the splitting of single elements in-line:

```
Inductive context_split : env A -> env A -> env A -> Prop :=
  | split_nil : context_split nil nil nil
  | split_left E E1 E2 v
            (SplitPre : context_split E E1 E2) :
            context_split (v :: E) (v :: E1) (None :: E2)
  | split_right E E1 E2 v
            (SplitPre : context_split E E1 E2) :
            context_split (v :: E) (None :: E1) (v :: E2).
```

Although for the purposes of forward reasoning both definitions are equally convenient, when performing inversion on terms of type `context_split E E1 E2` it is often useful to be able to abstract over the splitting of individual elements, particularly as this means there are less cases generated by the inversion.

For example, in the proof of `insert_none_split_backwards`, using context splitting on single elements saves duplicating or creating automation for the main part of the proof. The statement of the lemma is:

```
Lemma insert_none_split_backwards : forall A (E : env A) E1 E2 x,
  context_split (raw_insert x None E) E1 E2 ->
  exists E1' E2',
    E1 = raw_insert x None E1' /\
    length E1' = length E /\
    E2 = raw_insert x None E2' /\
    length E2' = length E /\
    context_split E E1' E2'.
```

Intuitively, this lemma states that if we have inserted `None` into the typing context at the position for variable `x`, then the two environments resulting from the split of this environment must also contain `None` at position `x`. The proof is by case analysis on whether or not the variable inserted is past the end of the environment (`x >= length E`). In the case where it *is not* past the end, and `x` is greater than 0, we reach a goal of the form:

```
exists X1 X2 : env A,
  e1 :: E1' = raw_insert (S x') None X1 /\
  length X1 = S (length E') /\
  e2 :: E2' = raw_insert (S x') None X2 /\
  length X2 = S (length E') /\
  context_split (e :: E') X1 X2
```

Here, we have `E = e :: E'`, `E1 = e1 :: E1'`, `E2 = e2 :: E2'`, `x = S x'` relative to the statement of the lemma.

Without a `split_single` definition, inversion of `context_split (e :: raw_insert x' None E') (e1 :: E1') (e2 :: E2')` forces us to consider the cases where `e1 = None, e2 = Some t` and `e1 = Some t, e2 = None` separately. In this proof, this isn't

an important distinction, because we really just need to apply the inductive hypothesis to obtain `E1''` and `E2''` such that `E1' = raw_insert x' None E1''` and `E2' = raw_insert x' None E2''`. We can then instantiate `X1` and `X2` with `exists (e1 :: E1''), (e2 :: E2'')` and knock over the remaining goals with facts from the proof context. This all works with a `split_single` definition, but without one, the two cases have to be handled with two different calls to the `exists` tactic; requiring duplication of proof script, or custom Ltac to parametrise the calls.

Hence, `split_single` is a more general and useful way for dealing with context splitting, as it allows us to prove goals of the form `context_split (e :: E) (e1 :: E1) (e2 :: E2)` without knowing the exact values of `e`, `e1` and `e2`. If the exact values *are* required for a proof, they can still be considered by performing inversion on the `split_single e e1 e2` fact.

3.7.2 Properties of Context Splitting

We prove several basic properties of context splitting, such as commutativity, as well as lemmas motivated by the proof of soundness for the linear lambda calculus.

Length If $E = E_1 \circ E_2$, then the lengths of all three contexts are the same.

Commutativity

If we can split $E = E_1 \circ E_2$, then we can also split $E = E_2 \circ E_1$. In Coq, proof is by a straight-forward induction on the structure of the `context_split`, and depends on a similar commutative property of `split_single`.

```
Lemma split_commute : forall A (E : env A) E1 E2,
  context_split E E1 E2 -> context_split E E2 E1.
Proof with boom.
  intros A E E1 E2 Split.
  induction Split...
Qed.
```

Associativity If $E = E_0 \circ (E_1 \circ E_2)$ then $E = (E_0 \circ E_1) \circ E_2$.

In Coq we need an existential to express the existence of a context that splits into E_0 and E_1.

```
Lemma split_assoc : forall A (E E0 E1 E2 E12 : env A),
  context_split E E0 E12 ->
  context_split E12 E1 E2 ->
  (exists E01, context_split E E01 E2 /\ context_split E01 E0 E1).
```

The justification for calling this operation associativity is the view of $(\circ)$ as a *partial* binary operator for combining contexts. From this view, context splitting is a partial commutative monoid, with `repeat n None` as the identity element. Another name for this structure is a *separation algebra* (assuming that $(\circ)$ is also cancellative, which it is). Further discussion of this is given later when evaluating our approach, in §4.3.

4-way Splits

In his proofs about uniqueness typing Edsko de Vries [dVPA07, dV08] establishes all of these properties, but proves the associativity lemma using a 4-way split lemma like this: $E = (E_{1a} \circ E_{1b}) \circ (E_{2a} \circ E_{2b}) \longrightarrow E = (E_{1a} \circ E_{2a}) \circ (E_{2a} \circ E_{2b})$. In addition to associativity being provable from this lemma, the opposite is also true, and we prove a version of this lemma called `split_swap` using a few simple applications of commutativity and associativity.

Rotation The proof of preservation requires a lemma of the form: $E = E_0 \circ (E_1 \circ E_2) \Rightarrow E = E_1 \circ (E_0 \circ E_2)$, which is provable using the other lemmas above.

Emptiness Lemmas about empty contexts as the identity element for context splitting are provable using a few lines of proof script. For details, see `Linear/Context.v` in the accompanying source code (§A.4).

3.7.3 Lemmas Required for Soundness

In order to prove soundness, several lemmas about the interaction between inserts and context splitting were required. Although many of the lemmas are conceptually simple, some required significant effort and represent a large portion of the work involved in the proof of soundness for the linear lambda calculus. Many lemmas also involve the insertion of `None` values into contexts, which is a consequence of the strengthened induction required to prove the substitution lemma (see the next section on substitution, §3.9.1).

The first, and perhaps most difficult to prove, was `insert_none_split_backwards`, discussed previously in the section about `split_single`. We attribute the difficulty in proving the lemma to the fact that we initially lacked ways to express many of the intuitive reasons for the lemma's truth, described below. With the right definitions and lemmas in place, the proof became quite straight-forward.

DbLib's `insert` function behaves in two different ways depending on the relationship between the variable `x` being inserted and the length of the existing environment, `E`. If `x < length E`, then the type for `x` is inserted *between* existing elements, with subsequent elements being shunted along. Conversely, if `x >= length E`, then the type of `x` is inserted *after* existing elements, with the intervening space padded by `None` values. These two cases are quite different to reason about, and any attempt at a direct inductive proof on `x` or one of the environments inevitably leads to wanting to know which of the two cases one is considering. For this reason, the first step of our proof is to split on the comparison between `x` and `length E`.

To handle the case where `x >= length E` and padding `None` values are inserted, we note that the values past the end of the old environment will all be `None`. Equipped with a simple definition of a `repeat` function on lists, and some lemmas about list append, the proof of this case is simple. Determining the right abstractions is the hardest part.

```
Lemma insert_none_def : forall A x (E : env A),
  x >= length E ->
  raw_insert x None E = E ++ repeat (S (x - length E)) None.

Lemma split_app : forall A (E : env A) E1 E2 n,
    context_split (E ++ repeat n None) E1 E2 ->
    exists E1' E2',
      E1 = E1' ++ repeat n None /\
      E2 = E2' ++ repeat n None /\
      context_split E E1' E2'.
```

The other case where the new `None` value is inserted in between existing elements is handled by an induction on `x`, which is made simpler by knowing the bound on `x`, i.e. `x < length E`. For example, it absolves us from having to deal with the case where the environment is empty, which proved to be a nuisance in early versions of the proof. Particularly as `length (tl E1) = length (tl E2)` does not imply that `length E1 = length E2` as one may expect outside a total programming language.

A few other lemmas about context splitting and interactions with DbLib's `insert` were required for the proof of soundness. They can be found in the accompanying proof sources, see §A.4. In general, the lemmas involving existentials required more cunning to prove, and subsequently more cunning to apply, as discussed in §4.4.2.

3.8 Progress

With typing rules and small step semantics established we can state the progress lemma which contributes to the proof of type soundness.

```
Theorem progress : forall E e t,
  is_empty E ->
  E |- e ~: t ->
  (exists e', step e e') \/ value e.
```

This theorem states that any closed term e that is well-typed can either be stepped using the small step semantic relation, or is already a value and cannot be evaluated further.

Proof is by induction on the typing derivation `E |- e ~: t`. Unlike the proof of preservation presented in the next section, the proof of progress requires very few supporting lemmas about context splitting or substitution. Most of the cases in the induction require reasoning about the interaction between typing and term structure, which can be handled by simple inversions. For example, one of the supporting lemmas states that any value assigned a function type under an empty environment must necessarily be a λ-abstraction:

```
Lemma fun_value_is_abs : forall E e t1 t2,
  is_empty E ->
  E |- e ~: TyFun t1 t2 ->
  value e ->
  (exists e', e = TAbs t1 e').
```

Given that most of the verification effort was expended reasoning about context splitting, substitution and other lemmas required for preservation, the effort required to prove progress represents a relatively small fraction of the total effort. The fact that progress and preservation each form "half" of the soundness proof does not imply that the difficulty of proving soundness is split evenly between them.

3.9 Preservation

The preservation lemma states that if a well-typed closed term e can take a step to another term e', then e' is also well-typed. In other words, the type of a term is *preserved* as it is evaluated in accordance with the small-step semantics.

```
Theorem preservation : forall E e e' t,
  is_empty E ->
  E |- e ~: t ->
  step e e' ->
  E |- e' ~: t.
```

Proof is by induction on the stepping of e to e', `step e e'`. In the three cases that result from the three stepping rules, the two for applications are proved directly from the inductive hypothesis. The remaining case for β-reduction involves interaction between substitution, typing and context splitting, and is proved via a supporting *substitution* lemma.

3.9.1 Substitution

The substitution lemma states that the result of a substitution is well-typed if the terms involved are well-typed. With substructural typing, we must also consider the supply of free variables to both terms using context splitting, so the lemma takes the form:

$$\frac{E_1 \vdash e_1 : \tau_1 \quad E_2, x : \tau_1 \vdash e_2 : \tau_2}{E_1 \circ E_2 \vdash e_2[e_1/x] : \tau_2} \text{ Substitution}$$

Substitution lemmas similar to this are common in proofs of similar complexity, as in *Software Foundations* [PCG+15] and the examples accompanying DbLib. In Software Foundations, a weakening lemma is used in the proof of the substitution lemma, but with substructural typing this technique is unavailable.

In Coq the lemma is:

```
Lemma substitution: forall E2 e2 t1 t2 x,
  insert x t1 E2 |- e2 ~: t2 ->
  forall E E1 e1, E1 |- e1 ~: t1 ->
  context_split E E1 E2 ->
  E |- (subst e1 x e2) ~: t2.
```

Proof is by *dependent induction* on the judgement `insert x t1 E2 |- e2 ~: t2`. Dependent induction allows us to take into account the fact that the environment is `insert x t1 E2` rather than an unadorned variable (e.g. E). This is achieved by replacing instantiated variables with general ones, and then adding constraining equalities. In this case, the only instantiated variable is `insert x t1 E2`, which dependent induction will replace by a new universally quantified variable E_{new} and the equation $E_{\text{new}} =$ `insert x t1 E2`. With the goal in the form described the dependent induction tactic then applies the induction principle for `has_type` to generate sub-goals for each of the cases, whilst preserving the newly added equality constraints. Coq's dependent induction is based on Conor McBride's `BasicElim` tactic [McB00] which makes use of Conor's humorously named "John Major" heterogeneous equality (`JMeq`). Heterogeneous equality requires the addition of an axiom, but our use of it is restricted to the proof-of-concept and it isn't required to use the library.

Simple inversion removes the cases that are absurd due to the `insert x t1 E2` environment, leaving three cases for variables, λ-abstractions and applications. The variable case is handled by some straight-forward reasoning about empty contexts, and requires no new supporting lemmas, while the other two cases form the motivation for several of the lemmas about inserts and context splitting.

The λ-abstraction case requires a proof that the term being substituted is well-typed under the environment for the abstraction sans binder, i.e. `(None :: E1) |- shift 0 e1 : t1`. This motivates `typing_insert_none`, which in turn motivates `insert_none_split`. Although `typing_insert_none` is more general in that it allows us to prove facts of the form `raw_insert x None E |- e : t` and not just `raw_insert 0 None E |- e : t`, generalising for all x makes related lemmas easier to prove by enabling induction on x. This technique is sometimes referred to as *inductive loading*.

The application case requires that one side of the application be well-typed without referring to the substitution variable x. This motivates the following lemma, `typing_insert_none_subst`:

```
Lemma typing_insert_none_subst : forall E e x junk t,
  raw_insert x None E |- e ~: t ->
  E |- subst junk x e ~: t.
```

This in turn motivates the rest of the lemmas about inserting none into a typing context, including the difficult to prove `insert_none_split_backwards` – discussed above.

To prove this lemma, DbLib was extended with a *lowering* operation that is conceptually inverse of lifting. In the case where variables are lowered by 1 we call the operation *unshifting*, by analogy with lifting by 1 (shifting). The lemma is proved by establishing that a similar lemma holds for `unshift x e`, and an equivalence of `unshift x e` and `subst junk x e` when `x` does not appear free in `e` (see `contains_var`).

The addition of lowering to DbLib was the only substantial change required for our proof of soundness for PLLC. A few other minor changes to use `raw_insert` instead of `insert` in some lemmas were also made. Although the lowering operation might be useful upstream, further work is required to unify it with lifting to reduce the maintenance burden for DbLib [Spr16].

3.10 Summary

Our Coq formalisation of PLLC was able to make use of universal lemmas about emptiness and context splitting developed as part of this thesis. These lemmas represent a significant portion of the effort required to establish type preservation for PLLC. Further, the formalisation was able to make successful use of François Pottier's DbLib library, for substitution operations and basic actions on typing environments. The creation of a generic library for context splitting provides a positive answer to our primary research question about exploiting commonalities in formalisations of languages with substructural typing. The extent to which the library is applicable to other formalisations is discussed in the next chapter.

Chapter 4

Evaluation

In this section we assess the quality of the work according to the Evaluation Framework of Section §2.14. We discuss the success of the library according to its conceptual goals, primarily by outlining how a formalisation of L^3 may make use of the library. Further, we evaluate the library's implementation according to the implementation goals and discuss possible improvements.

4.1 Generality and Applicability

According to the conceptual goals of our evaluation framework, we would like the library to be applicable to the formalisation of languages more complex than the linear lambda calculus. In concrete terms, this means that the context splitting operation provided by the library should be useful in constructing syntactic proofs of soundness for some class of languages with substructural typing. We argue that this class of languages includes those based on DILL – which could act as a foundation for further formalisations.

Extending the proof-of-concept LLC proof to a complete formalisation of DILL would require the addition of an intuitionistic context, product types and bang types. The paper for DILL [Bar96] includes proofs of substitution lemmas similar to the one used in the proof of preservation for our LLC, which suggests that it would admit a complete

proof of soundness in a syntactic style. This is in contrast to several earlier systems based on linear logic, which Philip Wadler showed *do not* have substitution lemmas [Wad91]. Wadler's result is our primary motivation for using DILL as a foundation, motivated further by Pottier's success with DILL-based syntactic soundness proofs in SSHPS [Pot13a]. Interestingly, a draft version of the L^3 paper from 2001 cites Wadler's result as a reason to avoid a syntactic proof, but this claim is absent from the final paper [MAF05].

The proofs of progress and preservation for DILL could re-use much of the proof effort for PLLC. For example, the ($\otimes$-I) rule is similar to PLLC's existing ($\multimap$-E) rule, implying that inductive cases related to ($\otimes$-I) could be handled using the same library lemmas used for ($\multimap$-E). Further, the lemmas provided by the library about `insert` and context splitting would aid in reasoning about ($\otimes$-E) and (!-E), which both include augmented contexts in their premises.

Our approach may also be applicable to a modified version of Edsko de Vries' uniqueness type system [dVPA07, dV08]. Edsko's soundness proof is syntactic and makes use of an inductive context splitting relation that is identical to ours except that it allows non-unique (intuitionistic) types to be split to both sides. To make his system compatible with purely linear context splitting a stand-alone contraction rule could be added, in the style of Wadler's linear lambda calculus of 1993 [Wad93]. The contraction rule would provide the ability to duplicate non-unique values, which would previously have been provided by context splitting.

4.2 Towards a Coq Formalisation of L^3

The Linear Language with Locations, L^3, is a good candidate for assessing the applicability of our approach to languages beyond the linear lambda calculus. As argued in the Background chapter, its use of linear capabilities and shared pointers is typical of other modern calculi supporting destructive updates.

L^3 seems like it may be amenable to mechanical verification with our library because

like DILL and Wadler's linear lambda calculus, contexts are split in a purely linear way. Intuitionistic terms are handled by the **dupl** t and **drop** t primitives, rather than implicit or explicit contraction and weakening, which means the `context_split` predicate could be employed unaltered. However, as noted in the L^3 paper [MAF05], the operational semantics and typing rules are not set-up for a syntactic proof of soundness. As an example, consider the preservation lemma for the memory allocation primitive **new** v. Under the operational semantics the term steps unconditionally: $(\sigma, \textbf{new}\ v) \Rightarrow (\sigma \uplus \{l \to v\}, \ulcorner l, \textbf{cap} \otimes !(\textbf{ptr}\ l)\urcorner)$, hence we should have:

$$\frac{\Delta; \Gamma \vdash \textbf{new}\ v : \exists \rho.\ A \quad (\sigma, \textbf{new}\ v) \Rightarrow (\sigma \uplus \{l \to v\}, \ulcorner l, \textbf{cap} \otimes !(\textbf{ptr}\ l)\urcorner)}{\Delta; \Gamma \vdash \ulcorner l, \textbf{cap} \otimes !(\textbf{ptr}\ l)\urcorner : \exists \rho.\ A}$$

The premises are both true, by the rules (New) and (new), but the conclusion is unprovable. It might seem that we could apply the (LPack) rule, but this would require $l \in \Delta$, which is nonsensical because l is a location constant, not a location variable. Further, we can't type the pair **cap** $\otimes !(\textbf{ptr}\ l)$ because there are no typing rules for lone capabilities or pointers. To do a syntactic proof of soundness would therefore require altering L^3's typing rules. The authors of the L^3 paper suggest using *store typing*, as in Alias Types [SWM00], whereby constraints on the runtime store σ are expressed in the typing rules. In this case, reasoning about context splitting would likely only comprise a small fraction of the proof effort. Pottier's proof for SSPHS and the proof for λ^{rgnUL} are around 20,000 lines of Coq and Twelf code respectively, implying that the complexity of reasoning about these language features (destructive strong updates) increases proof complexity significantly. For comparison, our linear lambda calculus formalisation is around 1,300 lines of Coq code including the library for context splitting.

4.3 Further Work: Separation Logic

Conor McBride's work on linear dependent types (§2.9), and François Pottier's on SSPHS (§2.8.2) both make use of typing contexts that record the number of available

occurrences of each variable, and use concepts from separation logic to handle context splitting. This approach is strictly more general than our context splitting library, and a library based on these ideas would likely form a better foundation for general reasoning about typing contexts in linear languages. In this sense, our library is lacking in generality. Further, ideas from separation logic are cited as foundational in much of the literature surveyed, and could be exploited for multiple purposes, as Pottier did with SSPHS [Pot13a]. Indeed, work on creating generalised libraries for separation algebras has already been completed, in Coq [DHA09] and Isabelle/HOL [KKB12].

4.4 Quality of Coq Proofs

As identified in the Implementation Goals (§2.14.2) section of our evaluation criteria, we would like the library to be both **readable** and **maintainable**.

4.4.1 Case Analysis and Indentation

To separate different cases when performing induction and destructuring, the proofs make use of Benjamin Pierce's `Case` markers from Software Foundations [PCG+15]. Combined with indentation, these markers fulfil our desire to create readable proofs that are also resistant to corruption upon refactoring. The `Case` tactic ensures that the proof remains structured by failing if an existing case at the same level remains unproven, as can happen if an earlier tactic fails. The following example taken from the proof of `insert_none_def` demonstrates our usage of the markers and our indentation scheme:

```
induction x as [|x']; intros.
Case "x = 0".
  destruct E as [|e E'].
  SCase "E = []".
    rewrite raw_insert_zero...
```

```
SCase "E = e :: E'".
  solve by inversion.
```

If, for example, the tactic `rewrite raw_insert_zero...` fails to prove the goal for the case where `E = []`, then the `SCase "E = e :: E'"` tactic will fail and prevent the proof from proceeding, clearly signalling that the error lies in the previous case. The string arguments are uninterpreted but provide useful documentation.

The style of proof shown above was followed meticulously throughout the entire development, aiding both readability and maintainability. Excessive nesting of cases was also successfully avoided, with the deepest level of nesting being 3 sub-cases deep (an `SSCase`), occurring only once in the proof-of-concept formalisation.

An alternative to using Pierce's `Case` markers would have been to use Coq's *bullets*, which are available as part of Coq's core since version 8.4 [Coq15]. They function identically to the case markers except that documentation strings can't be embedded as case descriptions.

4.4.2 Avoiding Auto-Generated Variable Names

Avoiding references to automatically generated names is an important part of creating a readable and maintainable Coq proof. Generated names are subject to change between Coq versions, potentially rendering all pieces of proof script reliant on them in need of upgrading. Furthermore, updating the proof can be difficult if the exact values that the names were referring to have been forgotten, effectively requiring old goals to be solved anew.

Although there are several straight-forward techniques that can be used to avoid generated names, in practice we didn't manage to avoid them entirely, notably when using dependent induction.

Introduction Patterns

When performing induction it is often necessary to leave some hypotheses as premises of the inductive hypothesis. In order to avoid generating names for these hypotheses we followed a pattern whereby the `intros` tactic would be used to name every hypothesis, `generalize dependent` would be used to re-quantify the necessary variables, and then induction performed. This leads to hypotheses being re-introduced with the names used by `intros`, rather than generated ones. For example:

```
Lemma empty_lookup : forall A x (E : env A), is_empty E ->
  lookup x E = None.
Proof.
  intros A x E Empty.
  generalize dependent E.
  induction x as [|x'].
  Case "x = 0".
    intros. inversion Empty; auto.
  (* Proof continues... *)
```

Here the fact `is_empty E` is named `Empty` by `intros`, abstracted over, and then automatically re-introduced with the name `Empty`. This relies partly on Coq's name generator to remember the name from the first application of `intros`, but this is less fragile than relying on entirely automatic naming.

Named Constructor Arguments

A cosmetic variation of the above pattern for introductions could use named function arguments instead of an explicit `forall`. This is the approach taken for all inductive constructors, with the aim of generating unique names during inversion. For example, we can declare the `is_empty` predicate in two semantically equivalent ways:

```
(* With implicit argument names *)
Inductive is_empty {A} : env A -> Prop :=
  | is_empty_nil : is_empty nil
  | is_empty_cons : forall E, is_empty E -> is_empty (None :: E).

(* With explicit argument names (preferred) *)
Inductive is_empty {A} : env A -> Prop :=
  | is_empty_nil : is_empty nil
  | is_empty_cons E (EmptyTail : is_empty E) : is_empty (None :: E).
```

With the second variant, when inverting a fact of the form `is_empty E` the fact about the environment's tail will be named `EmptyTail` if that name isn't already taken. In practice this was found to be quite effective, as most proofs required only a single inversion per data-type. In cases where more control over naming is needed, names can be provided to the tactic, as discussed in the next section.

Named Destructuring

Named destructuring allows names to be provided to tactics like `induction`, `destruct` and `inversion` which would otherwise generate names automatically. For simple cases like induction on natural numbers and lists, we found it to be highly effective, e.g. `induction n as [|n']`. However, destructuring large chains of existentials, conjunctions and disjunctions in this manner quickly becomes unwieldy.

For example, our proof of soundness for PLLC contains a line: `destruct AppPreSplit as [E1' [E2' [? [? [? [? ?]]]]]]`. This is to the detriment of both readability and maintainability, in particular because changes to the lemma that creates `AppPreSplit` would require digging into several layers of brackets to ensure the names are correct and that the chain of conjunctions is fully decomposed. This is an area where the library falls short of the evaluation criteria. Further work could seek to eliminate maintenance burdens such as these, possibly using Arthur Charguéraud's improved tactics library, `LibTactics` [Cha16].

Dependent Induction

Dependent induction is used at one point in the proof of soundness for PLLC, as discussed in §3.9.1. Unlike the regular induction tactic, Coq's dependent induction tactic doesn't provide a way to explicitly name the variables introduced by the inversion – there is no `dependent induction x as [..]`. In our proof, we fall back on using the auto-generated names, which is less than ideal. We experienced the fragility of this approach when refactoring, which demonstrates the utility of our other work to avoid generated names.

4.4.3 Automation and Repetition

We found that repeated sections of proof script could be de-duplicated in one of two ways: either by writing a lemma to encapsulate the common truth, or by using Coq's tactic language (Ltac) to repeat the same steps of reasoning. Of these, we found writing a lemma to be preferable as we found Ltac code more difficult to debug. In this section we discuss the efficacy of the automation employed by our proofs, and suggest possible improvements.

In his book *Certified Programming with Dependent Types* [Chl16], Adam Chlipala advocates:

"The more uninteresting drudge work a proof domain involves, the more important it is to work to prove theorems with single tactics." "I like to say that if you find yourself caring about indentation in a proof script, it is a sign that the script is structured poorly."

This is contrary to many of the steps taken by our proof to remain readable, including the use of case markers and indentation discussed above. This is primarily because constructing a proof in Chlipala's style is easiest when following his approach from the outset. Our proof began in a primarily manual style, with modest uses of the `auto` tactic with explicit hint lemmas. Mid-way through the development a hint database

was created with an accompanying specialised version of `auto` called `boom`.

Adapting the proof to use `boom` did remove repetition in some places, but was quite time-consuming for the savings achieved. The auto tactic works by repeatedly calling the `apply` tactic with lemmas from the hint databases and the current set of hypotheses. Unfortunately, many manual proofs are not structured as a straight-forward series of lemma applications, which means significant effort is often required to adapt proofs for use with auto. In our proofs, rewrites and inversion were the most significant barriers to automation with auto because they don't map cleanly onto stand-alone lemmas. To simulate a rewrite with a lemma requires restating the goal before and after the rewrite, which leads to a lemma for each goal and rewrite-rule pair. Similarly, the outcome of an inversion is usually a set of equalities about the components of the term being inverted, which can be used in a myriad of ways.

That said, the number of rewrites in our proof-of-concept is exacerbated by our choice to keep DbLib's definitions *opaque*. By default, DbLib exports the functions for lookups and inserts as opaque to the Coq simplifier. This means that an explicit rewrite is required every time a lookup or insert needs simplification – the `simpl` tactic has no effect. DbLib makes the definitions opaque by default to prevent *"fragile simplifications"*, and it is possible to selectively make definitions transparent again. If starting the proof again from scratch or reworking the proof to be more automated, we believe it would be best to make the definitions transparent for the whole project, with opt-in opacity where it is required to deal with odd simplification behaviour. This would remove rewrites and make `auto` applicable in more places.

As well as `auto`, Coq also includes an `autorewrite` tactic for repeatedly rewriting using a collection of rewrite rules. Use of this tactic could be investigated as an alternative to making definitions transparent, although it wouldn't solve the problem of automatically simplifying hypotheses, which we get for free with transparent definitions and the simplify-everywhere tactic, `simpl in *`.

Although automating proofs extensively reduces proof-effort and therefore makes larger developments feasible [Chl16], it is not without downsides. We found debugging Ltac

code quite difficult and labour-intensive. For example, Ltac's semicolon operator which makes it possible to "pipe" the results of tactics into each other, simultaneously makes debugging the middle of a pipeline difficult. The debugging mode for Ltac (`Set Ltac Debug`) doesn't show intermediate results in semicolon pipelines and also isn't available in CoqIDE or Vim. This leaves one with no option but to break the pipeline apart into individual tactic applications. If a tactic is failing in only a few branches this also requires the temporary use of the `admit` tactic to navigate to the failing cases. Once the error is found and fixed, one then has to glue the tactics back together. This debugging experience is unarguably less than ideal, and is an inevitable consequence of automating heavily using current tools. Our proof walks the line between difficult-to-debug Ltac code and repetitive manual proof, erring slightly on the side of manual proof. Further work could develop improved debugging facilities for Coq tactics, possibly using a graphical interface to convey the branching into different cases.

4.5 Summary of Evaluation

In summary, the library partially satisfies our evaluation criteria for quality of concept and implementation. Conceptually, the library is a success in that it aided us in the formalisation of PLLC. However, PLLC is extremely simple, and consideration of more complex languages like L^3 and SSPHS reveals that the library isn't general enough to be useful in the formalisation of extensions to the linear lambda calculus involving strong destructive updates. That said, we have argued that it is general enough to be of use in a proof of soundness for DILL, and there may be ways to construct compatible new languages with the same features as L^3. Further, it may be possible that reasoning about context splitting as we have defined it could be used in a non-syntactic proof of soundness, possibly following the semantic interpretation of types for L^3.

From an implementation perspective, the library and proof-of-concept are rather satisfactory. As discussed above, the fragility of automatic variable naming was avoided in all but one case involving the use of dependent induction. Readability of the proofs was simplest to achieve, through case markers and indentation, although the result

may still not satisfy Chlipala. More automation and the use of third-party convenience tactics are two areas identified as areas of slight deficiency. It's possible that with these improvements the development would also have been less time consuming.

Chapter 5

Conclusion

We have identified context splitting as a common trait of languages with substructural typing, and have created a Coq library for context splitting, building upon DbLib. This library aided in a syntactic proof of type soundness for a simple variant of the linear lambda calculus, and we are confident that it could be scaled to handle all of Dual Intuitionistic Linear Logic.

Overall we were able to achieve a satisfactory level of quality and expressiveness in the mechanised Coq proofs, although some deficiencies in the use of automatic variable naming and automation could be improved. Further work could look at using custom tactics for dealing with these problems in Coq, or address the more fundamental challenge of building readable and maintainable facilities for proof automation. Following Adam Chlipala's mostly-automated philosophy from the start would be beneficial from an automation standpoint, whilst also solving the issue of variable naming as a side-effect.

Although it was initially hoped that the context splitting library would be useful in the formalisation of more advanced languages, this is probably unrealistic. The first reason for this is that more advanced languages tend to involve type system extensions that obsolete context splitting as we have defined it, or contain so many other features that context splitting only makes up a small fraction of the overall proof effort. Secondly, the

approach to context splitting whereby unavailable values are entirely erased is subsumed by an approach based on separation algebras whereby the number of occurrences of a variable is tracked in the typing context. This approach is used to great effect by Pottier for SSPHS [Pot13a], and McBride for a fusion of linear and dependent typing [McB16].

Pottier's DbLib library for reasoning about de Bruijn indices was found to be highly effective at abstracting over the details of capture-avoiding substitution. Our proofs were able to make effective use of the library by defining only a few language-specific functions. DbLib's environment type, which is not used in Pottier's own work, was also found to be adequate for use with our simple linear language. Some minor additions were made to DbLib while constructing the proof of soundness for PLLC, and further work is required to integrate them into the upstream repository. Alternatively, the proof for PLLC could be restructured so as not to rely on lowering, which was the main addition to DbLib.

In a broad sense, the literature on substructural typing has reached a point of maturity where the creation of practical tools is becoming feasible. The next generation of operating systems, device drivers and language run-times can hopefully be built on the foundation of formal verification. Projects like Adam Chlipala's Bedrock [Chl11] and Cyclone [GHJM05] are already a move in this direction. Further work could build a compiler for a substructural language based on Pottier's SSPHS [Pot13a], with accompanying verification tools. Cross-over with the impressive world of C verification also seems like it would be productive.

Further, Mozilla's Rust programming language represents an opportunity for substructural typing to break into the mainstream. Although its semantics are yet to be formally specified, the foundations of Cyclone and the other systems surveyed could possibly be adapted for this purpose. Considering this alongside the possibility of verification tools built on linear dependent types as described in the work of McBride [McB16], these are surely exciting times for substructural type systems.

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 24

[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 29

[Bar96] Andrew Barber. *Dual Intuitionistic Linear Logic*. 1996. pages 5, 9, 53

[BPP14] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. *Submitted for publication*, 2014. pages 23, 28, 30

[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 27

[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 24

[Cha16] Charguéraud, Arthur. LibTactics, and TLC: a non-constructive library for Coq. http://www.chargueraud.org/softs/tlc/, 2016. pages 29, 59

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *ACM SIGPLAN Notices*, 46(6):234–245, 2011. pages 24, 65

[Chl16] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2016. pages 60, 61

[Coq15] Coq Development Team. The Coq Proof Assistant v8.4pl6: Reference Manual. https://coq.inria.fr/distrib/V8.4pl6/files/Reference-Manual.pdf, 2015. pages 57

[CS12] Pierre Castéran and Matthieu Sozeau. A gentle introduction to type classes and relations in coq. Technical report, Citeseer, 2012. pages 38

[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 27

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. *Higher-order abstract syntax in Coq*. Springer, 1995. pages 27

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W Appel. A fresh look at separation algebras and share accounting. In *Programming Languages and Systems*, pages 161–177. Springer, 2009. pages 56

[dV08] Edkso de Vries. *Making Uniqueness Typing Less Unique*. PhD thesis, Trinity College Dublin, 12 2008. pages 19, 46, 54

[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 19, 20, 29, 30, 46, 54

[FM04] Matthew Fluet and Greg Morrisett. Monadic regions. In *ACM SIGPLAN Notices*, volume 39, pages 103–114. ACM, 2004. pages 21

[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 21, 30

[GHJ+15] Dan Grossman, Michael Hicks, Trevor Jim, Greg Morrisett, and Nikhil Swamy. Cyclone the language, 2015. pages 21

[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 21, 65

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. pages 2, 5, 11

[GLAK14] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. *SIGPLAN Not.*, 49(6):429–439, June 2014. pages 4

[Gun92] Carl A Gunter. *Semantics of Programming Languages*. MIT Press, 1992. pages 12

[ISO11] ISO. Information technology – Programming languages – C. ISO 9899:2011, International Organization for Standardization, Geneva, Switzerland, 2011. pages 1

[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 21

[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

[KKB12] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *Interactive Theorem Proving*, pages 332–337. Springer, 2012. pages 56

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. pages 4, 25

[MAF04] Greg Morrisett, Amal Ahmed, and Matthew Fluet. The Linear Language with Locations: Technical Report TR-24-04. Technical report, Harvard University, 2004. pages 18

[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 13, 30, 54, 55

[McB00] Conor McBride. Elimination with a motive. In *Types for proofs and programs*, pages 197–216. Springer, 2000. pages 51

[McB16] Conor McBride. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, chapter I Got Plenty o' Nuttin', pages 207–233. Springer International Publishing, Cham, 2016. pages 23, 31, 65

[Moz15] Mozilla Resarch. The Rust Programming Language. https://rust-lang.org/, 2015. pages 4, 30

[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 24

[Nor98] Michael Norrish. C formalised in HOL. Technical report, University of Cambridge, 1998. pages 3

[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 26, 36, 40, 50, 56

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. pages 12, 27, 28, 72

[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 21, 22, 28, 30, 54, 56, 65

[Pot13b] François Pottier. The `dblib` library for de Bruijn indices in Coq. https://github.com/fpottier/dblib, 2013. pages 28

[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 21

[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 30

[Spr16] Michael Sproul. Pull request for DbLib: Implement a 'lowering' operation. https://github.com/fpottier/dblib/pull/5, 2016. pages 52

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Programming Languages and Systems*, pages 366–381. Springer, 2000. pages 30, 55

[TP11] Jesse A Tov and Riccardo Pucella. Practical Affine Types. In *ACM SIGPLAN Notices*, volume 46, pages 447–458. ACM, 2011. pages 11

[Twe08] Twelf Project. Twelf Wiki - Higher-order abstract syntax. http://twelf.org/wiki/Higher-order_abstract_syntax, 2008. pages 26

[Wad91] Philip Wadler. There's no substitute for linear logic. 1991. pages 54

[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 9, 54

[WF94] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994. pages 12

Appendix

A.1 PLLC Syntax in Coq

To define the terms of PPLC we must first define the set of types. The reason for this is that lambda abstractions are explicitly annotated with the type of their parameter in order to avoid type-inference when writing proofs. A lambda abstraction $(\lambda x : \tau.e)$ is represented by the Coq expression `TAbs` τ `e`, which hides the name of the binding variable x through the use of de Bruijn indices. The inductive definition for types is:

```
Inductive ty : Set :=
  | TyUnit
  | TyPrim : String.string -> ty
  | TyFun : ty -> ty -> ty.
```

Now, with PLLC's types defined, we can define the set of terms:

```
Inductive term : Set :=
  | TUnit
  | TPrim : String.string -> term
  | TVar : nat -> term
  | TAbs : ty -> term -> term
  | TApp : term -> term -> term.
```

Variables are represented by the `TVar` constructor which takes a de Bruijn index representing the variable and constructs a `term`. Later when defining the typing judgement we will see that a variable `TVar i` is well-typed only if the typing context contains a type at index i.

In order to state the progress and preservation lemmas, Coq also needs an idea of which terms are considered *values*, i.e. those terms in a form that can not be simplified further. For this, we use an inductive predicate `value t` with the following definition:

```
Inductive value : term -> Prop :=
  | VUnit : value TUnit
```

```
  | VPrim : forall s, value (TPrim s)
  | VVar : forall x, value (TVar x)
  | VAbs : forall t e, value (TAbs t e).
```

Note that the type of `value` is `term -> Prop`. Given a `term`, `t`, a Coq term with type `value t` *witnesses* the truth of the proposition `value t`, which itself has type `Prop`. This is in contrast to `ty` and `term` which have type `Set`. Types in Prop are intended to represent proof terms, while those in Set are meant to represent data. We use Prop for all predicates to minimise friction with Coq's standard library, which provides definitions for basic logical connectives operating only on `Prop`s.

A.2 PLLC Typing Rules in Coq

Typing rules determine which terms are considered well-formed. We define an inductive predicate `has_type : (env ty) -> term -> ty -> Prop` so that `has_type E e t` is inhabited if the environment E determines $e : t$. In standard mathematical notation this is $E \vdash e : t$, which we mirror using the Coq notation `E |- e ~: t`.

```
Reserved Notation "E '|-' e '~:' t" (at level 40).

Inductive has_type : (env ty) -> term -> ty -> Prop :=
  | HasTyUnit E
      (UnitPre : is_empty E) :
      E |- TUnit ~: TyUnit
  | HasTyPrim E s t
      (PrimPre : is_empty E) :
      E |- TPrim s ~: TyPrim t
  | HasTyVar E x t
      (VarPre : is_empty E) :
      insert x t E |- TVar x ~: t
  | HasTyAbs E e t1 t2
      (AbsPre : (insert 0 t1 E) |- e ~: t2) :
      E |- TAbs t1 e ~: (TyFun t1 t2)
  | HasTyApp E E1 E2 e1 e2 t1 t2
      (AppPreSplit : context_split E E1 E2)
      (AppPreWT1 : E1 |- e1 ~: TyFun t1 t2)
      (AppPreWT2 : E2 |- e2 ~: t1) :
      E  |- TApp e1 e2 ~: t2

where "E '|-' e '~:' t" := (has_type E e t).
```

Here we see the definitions for emptiness and context splitting coming into play. In linear lambda calculus variables must be used exactly once, so a single variable `x` is well-typed only under an environment containing a type for `x` and nothing else, as captured by the `HasTyVar` rule. Similarly, primitive values must be typed under environments containing no free variables.

A lambda abstraction ($\lambda\hat{0} : \tau.\ e$) is well-typed if the body can be typed under an environment extended by the type for its binder, ($\hat{0} : \tau$). This is the only rule that requires the *input* context (`insert 0 t1 E`) to differ in length to the *output* context (`E`).

Function applications make use of the `context_split` operation to ensure that the variables used to type an application ($e_1 e_2$) are split between e_1 and e_2 without duplication. The rule also ensures that the type of the application (`t2`) is consistent with the type of the function being applied (`t1 -> t2`) and the type of the argument (`t1`).

A.3 PLLC Operational Semantics in Coq

The following inductive definition encodes $e \Rightarrow e'$ as `step e e'`.

```
Inductive step : term -> term -> Prop :=
  | StepAppAbs e e' v t
      (BetaPreVal : value v)
      (BetaPreSubst : subst v 0 e = e') :
      step (TApp (TAbs t e) v) e'
  | StepApp1 e1 e1' e2
      (App1Step : step e1 e1') :
      step (TApp e1 e2) (TApp e1' e2)
  | StepApp2 v1 e2 e2'
      (App2Val : value v1)
      (App2Step : step e2 e2') :
      step (TApp v1 e2) (TApp v1 e2').
```

The first rule, `StepAppAbs` is the familiar β-rule for evaluating the application of an abstraction to a value. It states that function application is equivalent to the substitution of the argument value for the bound variable in the body of the function. For substitution, note the use of DbLib's `subst` function.

We are using a *call-by-value* evaluation strategy, which means that only the outer-most reducible expressions (*redexes*) are reduced, and only when their argument is a value [Pie02]. Note that as a result there is no rule for stepping a λ-abstraction when its body is capable of stepping, as in: $e \Rightarrow e' \longrightarrow (\lambda\hat{0} : \tau.\, e) \Rightarrow (\lambda\hat{0} : \tau.\, e')$.

A.4 Complete Source Code

The complete source code for both the library and the proof of soundness for PLLC is available on GitHub.

https://github.com/michaelsproul/dblib-linear

The commit-hash for the master branch at the time of writing is:

`f4e6fb9becdaa208943ce0d180f2a30ef2f381de`