Clamp: Type Classes for Substructural Types

Edward Gan, advised by Greg Morrisett and Jesse Tov

A thesis presented to
the Department of Computer Science and
the Department of Mathematics

in partial fulfillment of the requirements for the degree of
Bachelor of Arts

Harvard University
Cambridge, Massachusetts

March 29, 2013

Abstract

The inner workings of most computer systems must manipulate both pure data and volatile resources. Numbers are usually an example of pure data, since they can be freely copied and used wherever they are needed. However, there are situations where files must be opened and closed, memory must be allocated and freed, and locks must have their ownership tracked while being passed carefully from thread to thread. Managing dynamic resources such as these is a perennial source of programmer frustration.

Substructural logics are a mathematical framework for reasoning about resources. Unlike standard logics, substructural logics limit where their assumptions can be reused or ignored. Thus, programming languages have been developed which adapt these logics into their type systems. These languages control the usage of resources by restricting where values can be reused or ignored.

In this thesis I present Clamp, a programming language with a substructural type system that is framed in terms of type classes. Type classes are a system for constraining types by the operations they support, and their integration with substructural types offers both theoretical and practical advantages. Clamp supports a variety of polymorphic substructural types as well as a powerful system of mutable references. At the same time, the Clamp core calculus remains simpler than ones in other substructural languages.

I have also implemented a type checker for Clamp in Haskell, and its design provides evidence that the substructural types in Clamp fit cleanly on top of standard type checking algorithms.

Acknowledgements

This thesis would not have been possible without the support I've received from a great many people.

My advisors, Greg Morrisett and Jesse Tov, have invested a tremendous amount of attention as I worked with them on this project. Greg Morrisett is a tireless source of wisdom and his warnings have saved me many times from nasty dead ends. His thirst for robust language design is infectious, and is the animating spirit in this thesis. Jesse Tov has been invaluable in helping me set the scope of this work. His standards of research are something I aspire to, and I cannot thank him enough for working through so many of my typing derivations.

The computer science department at Harvard has been a fantastic environment for learning. I would especially like to thank Harry Lewis for, among other things, imparting a love of machine models, and Steve Chong for giving me an appreciation for the intricacies of memory. Dan Grossman, currently at UWash, taught me to see the beauty in type soundness proofs.

My fellow classmates are a daily source of inspiration, and my friends who have brought unwarranted snacks and dragged me away from the computer have made the process of thesis writing not only productive but fun.

Finally, I am grateful to my parents. My mother's warmth and my father's conviction have been a great comfort in this endeavor as in many others.

Contents

Chapter 1

Introduction

Programming languages have made substantial progress in abstracting away physical constraints. Even in a low-level language such as C, one can write code that manipulates integers at will, reusing and discarding them as if they were pure mathematical objects rather than bits in the machine. However, it is hard to escape completely from issues of state and resource usage: programmers still routinely make mistakes in accessing memory after it has been deallocated, writing to a shared pointer without synchronization, sending data along a socket before acknowledgement, etc... It is difficult to write correct state-handling code partly because most compiler and type systems do not do a good job of statically tracking state.

In this thesis, I present a framework for developing programming languages with static state management. More specifically, I argue that by formulating *substructural types* in terms of *type classes* (explained later), one can design type systems to manage state in a way that is both theoretically elegant and convenient for programmers.

On their own, substructural types provide a way of controlling state by counting resource usages, but adding them to existing languages requires major additions to the type system and unfamiliar forms of programmer annotation. On the other hand, type classes were originally a mechanism for function overloading, but have become widely accepted for purposes ranging from generic programming to type level computation: their usability in theory and in practice is well established. In some sense, by encoding substructural types as instances of type classes, one can add substructural types to a language at marginal cost. Moreover, the interactions between type classes and substructural types make it easy to include rich state-aware datatypes such as *weak* and *strong references.*

To exhibit this, I have developed the CLAMP programming language, which brings together type classes, substructural types, and rich reference types into a coherent whole. In the remainder of this chapter, I describe the intuition behind substructural types and survey previous language designs that include substructural types. Then I introduce the concept of type classes and describe how they can be used to support substructural types.

Chapter 2 serves as an informal description of the Clamp programming language, structured as a tutorial and highlighting its support for substructural types alongside polymorphism and mutable references. In chapter 3, I describe an algorithm for inferring the substructural annotations that the Clamp type system expects, easing boilerplate burdens on the programmer while keeping the type system simple. Chapter 4 describes the Clamp type system in mathematical detail and presents a basic semantics to prove type soundness. Chapter 5 documents the development of the Clamp type checker as an extension of a Haskell type checker, and sketches a more fully-featured reference counting semantics for Clamp. Finally, I conclude in chapter 6 by summarizing the contributions of the thesis and pointing towards future work.

1.1 Substructural Types

Type systems enforce rules about the kinds of data that can be used at certain points of a program. They help programmers in writing correct code and compilers in generating efficient code [Harper and Morrisett, 1995]. However, while standard type systems can enforce data usage policies, they do little to help programmers manage resources, or more generally, protocols involving state. This shortcoming has inspired a host of research into tools such as Valgrind to detect incorrectly allocated memory, or static analysis tools such as SLAM [Ball and Rajamani, 2002] to check for driver protocol compliance.

Substructural type systems make it possible to control resources and state by restricting the number of uses of a value [Walker, 2005]. We will consider four classes of substructural types that classify values with different usage restrictions.

linear values that must be used exactly once

affine values that can be used at most once

relevant values that must be used at least once

unlimited values with no restrictions on their uses

One way of making sense of these four classes is to view them as a lattice from most restrictive to least restrictive as in figure 1.1.

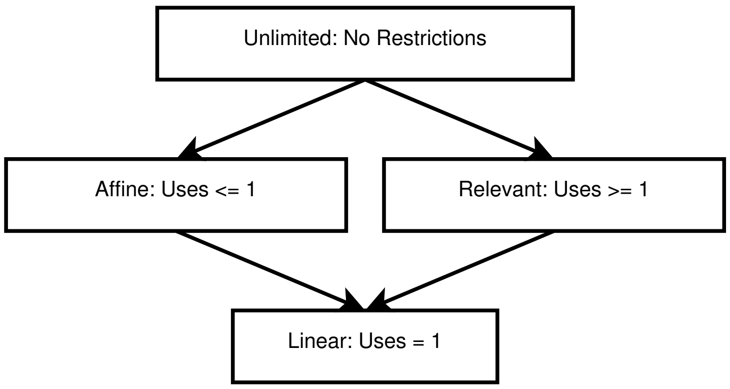

Figure 1.1: URAL Lattice

As examples of substructural types, consider a type system with distinct integer types distinguished by substructural specifiers. Given a linear integer type $\mathrm{int}^{\mathrm{L}}$ the function

$$\lambda\left(x : \mathrm{int}^{\mathrm{L}}\right) . \, x + x$$

would be ill-typed since once x is bound, it is used twice, while

$$\lambda\left(x:\mathrm{int}^{\mathrm{R}}\right).\ x+x$$

would be well-typed for a relevant integer type $\mathrm{int}^{\mathrm{R}}$. On the other hand,

$$\lambda\left(x:\mathrm{int}^{\mathrm{R}}\right).\ 5$$

would be ill-typed since x is ignored (used less than once) after it is bound.

By restricting usage in this way, one can enforce a variety of policies involving resources or state. These range from socket protocols to reader-writer locks, and include any protocol expressible as a DFA [Tov and Pucella, 2011, Mazurak et al., 2010]. For instance, if one assigns file handles a linear type, then one can ensure (assuming termination) that a handle will be closed exactly once, since as a value with linear type it must be either closed and eliminated or passed onto another function. Other metaphorical examples of substructural resources include gold coins, which are affine if one allows for losing coins but not reusing them as payment, and radioactive waste, which is relevant if one allows for spreading contamination but not ignoring it.

The file handle example is a simple illustration of the power of linear types, and it is useful to go into more detail here to motivate substructural types. In Walker [2005] the authors describe a design for a file I/O library which considers file handles as a linear type to ensure that they are opened and closed properly. In an OCaml-like syntax then, the interface to their library is reproduced below:

```
type file : linear

val open : string -> file option
val read : file -> string * file
val write : file * string -> file
val close : file -> unit
```

The `open` function generates linear file handles in a way which is opaque to the library user, while the `close` function consumes file handles. Since files are linear, once a file has been closed the same handle cannot be reused to further read, write, or close the same file. The library still allows for the repeated reading and writing because these two functions consume a file handle, but also return a fresh file handle for further access to a file. At runtime the library implementations of read and write may in fact return the same file handles they were passed, but statically they provide a way to thread the usage of linear file handles together.

Even more so than files, one of the most prevalent and precious resources on a computer is memory, and substructural types offer a host of possibilities for managing this resource as well. In a language with linear types, a linear value can be neither duplicated or discarded, so its value can be either deallocated or reused immediately after use [Wadler, 1991], eliminating the need for garbage collection on this value. Linear types can also be used to enforce proper copying and disposal of unlimited values, formalizing the development of reference counting systems with provably absent memory leaks [Cheney and Morrisett, 2003, Chirimar et al., 1996]. Though this does not extend to cyclic data structures, it simplifies memory management for a large class of functional, DAG-like structures.

When mutable references are added to the picture, substructural types provide further guarantees on aliasing and uniqueness. In the Cyclone programming language [Swamy et al., 2006] affine pointers for instance can be used to control initialization of memory cells. A more unified framework for substructural references is developed in λ^{URAL} where substructural pointers allow safe use of an operation called a *strong update* [Ahmed et al., 2005]. In most programming languages, type safe usage of pointers and mutable references require that the data stored in a reference cell maintain the same type as one updates it, so that no dereference comes to expect a value of a certain type when the type has changed in the meantime. The process of updating the contents of a reference to a new value with the same type is known as a *weak update*. Since C, like most languages, has a type system which only checks for weak updates, the following C code is ill-typed without an explicit cast because it attempts to perform a *strong update*:

```
char *cptr = 'a';
*cptr = 1;
```

On the other hand, if one can guarantee that no other thread of control has access to a reference cell at some point, it can be safe to change the very type of data stored in a reference as long as the static type change is tracked, and this is a strong update. Since affine and linear pointers cannot be shared, they provide a natural way to control the usage of strong updates.

In summary, substructural types allow programmers to safely manage resources and state, especially those involving memory and references.

1.2 Related Work

The development of languages with substructural types has been an active area of research, and we cannot hope to cover the wide expanse of related work in this area. In this section we will focus on a few illustrative examples, and try to classify related type system designs along one of three strands: those with "!" operators inspired by linear logic, those with qualifiers, and those with substructural kind systems.

1.2.1 Linear Logic

The first inspiration for substructural types comes from proof theory, with the development of *linear logic [Girard, 1987].* This logic was inspired by the analogy of causal reasoning in daily life. For instance, suppose \$1 is the cost of item A, and \$1 is also the price of item B. In this sense one might write $\$1 \multimap A$ and $\$1 \multimap B$ where $\multimap$ is the linear logic analogue of the "implies" relation $\rightarrow$. However, it isn't necessarily true then that one can use \$1 to simultaneously buy both item A and item B. In other words one cannot say from the above that $\$1 \multimap A \otimes B$ where $\otimes$ is the linear logic analogue of the "and" relation $\wedge$.

What one can say instead is that one could use \$1 to choose to buy *either* item A or item B. In the language of linear logic this is written $\$1 \multimap A \& B$ where & is the linear logic representation of a "choice". This is distinct from the classical "or" relation $\vee$ since classical disjunction doesn't allow one to choose which branch is true.

Given this intuitive description of $\multimap$ (linear implication), $\otimes$ (called "times"), and & (called "with"), we can write down some representative inference rules for linear logic in the form of the sequent calculus judgments $\Gamma \vdash A$, which can be read that the proposition A is true assuming the propositions in Γ.

Linear-Logic-Core

$$\textsc{LL-Id}\quad \frac{}{A \vdash A} \qquad \textsc{LL-TimesI}\quad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \textsc{LL-WithI}\quad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}$$

$$\textsc{LL-ImpI}\quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \textsc{LL-ImpE}\quad \frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

The key characteristic distinguishing it from classical intuitionistic logic is the fact that assumptions are treated as conserved resources: they must be discharged exactly once. Linear logic does not allow one to reuse or destroy the assumptions given in Γ. The fact that assumptions must be used at least once is enforced by the LL-Id rule: there can be no extraneous assumptions in a derivation. The restriction on using assumptions more than once is enforced by the combining of assumptions in rules such as the LL-TimesR rule. For instance, to derive the conjunction $A \otimes B$, one must assume both the assumptions used to derive A and those used to derive B. The same proposition may have to be assumed multiple times. Thus, in linear logic the context of assumptions Γ is a multiset of propositions.

A purely linear logic however is very weak, so to express standard classical propositions the "!" operator (often pronounced "of-course") is introduced. Propositions marked with "!" are intuitively "always true" and so are unlimited in the sense that unlike normal linear propositions, they can be reused or ignored.

Linear Logic Bang

$$\textsc{LL-Weak}\quad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \qquad \textsc{LL-Contract}\quad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \textsc{LL-Promote}\quad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A} \qquad \textsc{LL-Derelict}\quad \frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}$$

The act of ignoring an unlimited proposition in the LL-Weak rule is known as weakening, and the act of reusing an unlimited proposition in the LL-Contract rule is known as strengthening. These are called substructural operations because the involve manipulating the structure of the assumption context in ways that are usually implicit. Throughout this chapter, we will not treat the substructural exchange operation and consider contexts as multisets rather than lists.

The $!\Gamma$ notation is shorthand for a context whose propositions are all marked with a "!", so the LL-promote rule tells us that propositions can be marked with "!" when their assumption dependencies allow for it. The LL-Derelict rule allows us to ignore power of "!" at any point.

1.2.2 Linear Logic-inspired Languages

Guided by the Curry-Howard isomorphism, one can develop a term language and type system which mirrors the structure of linear logic [Bierman, 1993, Wadler, 1991, Abramsky, 1993]. One possible formulation adapted from Wadler [1991] and Bierman [1993] is given in figure 1.2 on the next page, where we give the typing judgment $\Gamma \triangleright M : A$ assigning a term M the type A under context Γ.

This type system is an example of a substructural type system because it restricts the substructural operations of weakening (i.e. ignoring variable binding assumptions) and contraction (i.e. reusing variable binding assumptions). In this case, the two operations are restricted to types annotated with "!".

This yields a relatively simple theory, but the "!" annotations are prevalent in types whenever one wishes to make use of unlimited values, and in an explicitly typed calculus (not the one given in figure 1.2) are also prevalent in the terms. In an implicitly typed calculus such as the one given in figure 1.2, full inference of "!" is impossible since one term can be assigned many incompatible types [Wadler, 1991]. For instance, the term $\lambda x.x$ can be assigned the types $A \multimap A$ or $!A \multimap A$ or $!A \multimap !A$, but has no most general type.

It is possible to infer a kind of principal "type scheme" for unannotated terms in this theory using *use-types*, which impose arithmetic constraints on the occurrences of "!" [Wadler, 1991], but these schemes can be cumbersome and ad-hoc, and their strengths are captured by the qualifier based languages discussed below.

1.2.3 Qualifier-Based Languages

Another approach to exposing substructural types is to break types into a *qualifier* and a *pre-type*. For instance, the system given in Ahmed et al. [2005] has the forms given in figure 1.3 on the facing page. Types τ have kind $\star$, and are broken down into a pre-type $\overline{\tau}$ of kind $\overline{\star}$ and a qualifier ξ of kind Q.

The qualifier (Linear, Unlimited, Affine, etc.) specifies once and for all what substructural properties a type possesses, while the pre-type specifies the base type. For instance, $^{\text{linear}}$int and $^{\text{affine}}$int are distinct types in this kind of system. This removes many of the ambiguities with the "!" operator, but requires more heavyweight machinery for polymorphism, since one must introduce verbose qualifier level polymorphism to support functions that can handle say both linear and affine integers [Walker, 2005, Ahmed et al., 2005]. For instance, a polymorphic pair constructor function might be assigned the following type if it is to work on arguments with different substructural properties:

Linear-λ Bang

$$\textsc{LLE-Id}\quad \frac{}{x : A \triangleright x : A}$$

$$\textsc{LLE-ImpI}\quad \frac{\Gamma, x : A \triangleright M : B}{\Gamma \triangleright \lambda x.M : A \multimap B} \qquad \textsc{LLE-ImpE}\quad \frac{\Gamma \triangleright M : A \multimap B \qquad \Delta \triangleright N : A}{\Gamma, \Delta \triangleright M\ N : B}$$

$$\textsc{LLE-TimesI}\quad \frac{\Gamma \triangleright M : A \qquad \Gamma \triangleright N : B}{\Gamma, \Delta \triangleright (M, N) : A \otimes B} \qquad \textsc{LLE-TimesE}\quad \frac{\Delta \triangleright M : A \otimes B \qquad \Gamma, x : A, y : B \triangleright N : C}{\Gamma, \Delta \triangleright \text{case } M \text{ of } x \otimes y \rightarrow N : C}$$

$$\textsc{LLE-WithI}\quad \frac{\Gamma \triangleright M : A \qquad \Gamma \triangleright N : B}{\Gamma \triangleright [M, N] : A\&B} \qquad \textsc{LLE-WithE1}\quad \frac{\Gamma \triangleright M : A\&B}{\Gamma \triangleright \textbf{fst}\ M : A} \qquad \textsc{LLE-WithE2}\quad \frac{\Gamma \triangleright M : A\&B}{\Gamma \triangleright \textbf{snd}\ M : B}$$

$$\textsc{LLE-Promote}\quad \frac{!\Gamma \triangleright M : A}{!\Gamma \triangleright M :!A} \qquad \textsc{LLE-Derelict}\quad \frac{\Gamma, x : A \triangleright M : B}{\Gamma, x :!A \triangleright M : B}$$

$$\textsc{LLE-Weak}\quad \frac{\Gamma \triangleright M : B}{\Gamma, x :!A \triangleright M : B} \qquad \textsc{LLE-Contract}\quad \frac{\Gamma, x :!A, y :!A \triangleright M : B}{\Gamma, z :!A \triangleright \{z/x\}\,\{z/y\}\,M : B}$$

Figure 1.2: Linear Lambda Calculus with "!"

λ^{URAL}-Types

$$\kappa ::= \mathrm{Q} \mid \overline{\star} \mid \star$$

$$\tau ::=^{\xi} \overline{\tau}$$

$$\overline{\tau} ::= \alpha \mid 1_{\otimes} \mid \tau_1 \otimes \tau_2 \mid \tau_1 \multimap \tau_2 \mid \forall\alpha : \kappa.\tau$$

$$\xi ::= \alpha \mid q$$

$$q \in \{\mathbf{U}, \mathbf{R}, \mathbf{A}, \mathbf{L}\}$$

Figure 1.3: λ^{URAL} Types

$$\mathit{pair} : \forall \xi_1 : \mathrm{Q}.\ \forall \tau_1 : \overline{\star}, \tau_2 : \overline{\star}.\ \left({}^{\xi_1}\tau_1\right)^{\mathbf{U}} \multimap \left({}^{\xi_1}\tau_2\right)^{\mathbf{U}} \multimap^{\xi_1} \left(\left({}^{\xi_1}\tau_1\right) \otimes \left({}^{\xi_1}\tau_2\right)\right)$$

Even with the annotations for qualifier polymorphism, this doesn't allow us to apply "pair" to two types with different qualifiers without either the complications of subtyping or a richer qualifier language. We might like to write a pair constructor function which assigns the pair the most general substructural qualifier it can, but this is not possible unless one expands the qualifier language to allow a type such as the one below:

$$\mathit{pair} : \forall \xi_1 : \mathrm{Q}, \xi_2 : \mathrm{Q}.\ \forall \tau_1 : \overline{\star}, \tau_2 : \overline{\star}.\ \left({}^{\xi_1}\tau_1\right)^{\mathbf{U}} \multimap \left({}^{\xi_2}\tau_2\right)^{\mathbf{U}} \multimap^{\xi_1 \sqcup \xi_2} \left(\left({}^{\xi_1}\tau_1\right) \otimes \left({}^{\xi_2}\tau_2\right)\right)$$

In summary, qualifier based systems are flexible but heavyweight.

Before moving on to kind based systems, it is worth mentioning another well known example of a substructural type system: the one found in the Clean programming language [Barendsen and Smetsers, 1996, Vries et al., 2008]. Clean supports *uniqueness types* which are similar to linear types but support subtyping in a different direction than that found in most linear type systems. Otherwise it fits within the framework of other qualifier based substructural type systems.

1.2.4 Kind-Based Languages

Recent languages such as Alms [Tov and Pucella, 2011] and F° [Mazurak et al., 2010] remove much of the overhead in qualifier based type systems by using distinct kinds (rather than qualifiers) to separate substructural types. They do this by observing that the decoupling between qualifiers and pre-types is often unnecessary. For instance, one rarely needs to use ${}^{\mathbf{U}}\mathrm{int}$ or ${}^{\mathbf{R}}\mathrm{int}$ types since integers are naturally thought of as unlimited mathematical quantities. On the other end of the spectrum, types such as ${}^{\mathbf{U}}\mathrm{file}$ should be illegal if files are always linear types. Instead, in these systems one sets aside distinct kinds for distinct substructural classes, so that for instance file would be a type which requires no annotations but can only be assigned the kind $\kappa = \mathbf{L}$.

By introducing subtyping, subkinding, and dependent kinds, Alms is able to achieve a striking amount of polymorphism and code reuse with very low programmer overhead [Tov and Pucella, 2011]. The issue with writing a polymorphic pair constructor functions is simple to resolve in Alms, since the pair type constructor $\otimes$ can be given a dependent kind, so that given a type $\alpha \otimes \beta$ one can derive the most general kind for the pair given the kindings of the components.

$$\frac{\textsc{Alms-K-Prod} \qquad \vdash \Gamma}{\Gamma \vdash (\otimes) : \Pi\alpha^+.\Pi\beta^+.\ \langle\alpha\rangle \sqcup \langle\beta\rangle}$$

However, relatively large amounts of metatheoretic machinery are required to support such a kind system, and it can be hard to extend kind based systems. For instance, Alms only includes affine and unlimited types, and if one wanted to add relevant and linear types, it would require adding additional kinds as well as the relevant subkinding rules. Combining these new kinds with the extensive kind systems already present in other languages requires even more work to incorporate the entire lattice.

One closely related area of research which we have not discussed centers around capability calculi [Crary et al., 1999]. The Vault programming language for instance includes a type system based on a capability calculus designed specifically to track resource state [DeLine and Fähndrich, 2001]. It does this by associating resources with basically linear "keys" which are kept distinct from normal types using a kind system. However, in Vault linearity is not exposed as a general type system feature and the design choice made is to restrict them to keys.

1.3 Type Classes

Type classes are a mechanism for supporting *ad-hoc polymorphism* in functional programming languages. The standard list "map" function is *parametrically* polymorphic because its type scheme $\forall\alpha,\beta.\,(\alpha \to \beta) \to \text{list }\alpha \to \text{list }\beta$ allows it to be instantiated with any concrete types α and β, while behaving in a way that is parametric with respect to (i.e. independent of) the instantiated types. On the other hand, ad-hoc polymorphic functions have implementations which depend specifically on the types of their arguments, and may not even support certain argument types. Ad-hoc polymorphism is synonymous with *function overloading.* For instance, numeric addition $+$ behaves differently depending on whether it is passed an integer or a floating point number, but is not defined over arbitrary function types.

Type classes are a mechanism for supporting ad-hoc polymorphic functions by assigning them a constrained type scheme $\forall\overline{\alpha_i}\,[P]\,.\tau$. Constrained type schemes can only be instantiated with types that satisfy the constraints P. In the case of type classes, these constraints take the form of membership predicates $K\ \tau$ specifying that τ must be a member of the *class* K. A type satisfies a membership predicate $K\ \tau$ when there exists an implementation for the functions associated with K on τ. Thus, different implementations can be used depending on the type which one uses to instantiate the type scheme.

As an illustrative example, the Haskell programming language includes an overloaded equality function "eq" whose type is $\forall\alpha\,[\text{Eq }\alpha]\,.\alpha \to \alpha \to \text{bool}$, meaning that eq can be applied to any argument whose type is a member of the Eq class. In Haskell, this type would be written:

```
Eq a => a -> a -> Bool
```

The Eq class is defined by a class declaration which specifies the collection of methods all members of a type must support. The class declaration is similar to an interface declaration in object oriented languages.

```
class Eq a where
  eq :: a -> a -> Bool
```

Finally*, instances* are provided for the Eq class which provide implementations of the equality function on various types, and thus also define the types under which the corresponding class constraint holds. Note that the definition of equality on pairs depends upon a definition of equality on its components, and thus the constraint Eq $\alpha \times \beta$ holds only when the Eq constraint holds on the components of the pair.

```
instance Eq Int where
  eq i1 i2 = int_compare i1 i2

instance (Eq a, Eq b) => Eq (a,b) where
  eq (p11,p12) (p21,p22) = (eq p11 p21) and (eq p12 p22)
```

1.4 Type Classes for Substructure

If we think of substructural types not in terms of usage counts but in terms of the substructural operations they support, then type classes become a natural system for encoding substructural types, and address many of the issues found in the three previous approaches to substructural type systems.

In Clamp, all variables are thus considered linear, and we provide explicit dup (contraction) and drop (weakening) operations that copy or ignore values, and that are mediated by typeclasses. For example, in this system relevant types would simply be those for which we define an instance for dup but not drop. [Wadler and Blott, 1989].

In the language of type classes, we incorporate substructural types through Dup and Drop classes which describe exactly the types which support the duplication and disposal operations respectively. In Haskell, the Dup and Drop operations might be defined as follows:

```
class Dup a where
  dup :: a -> (a,a)

class Drop a where
  drop :: a -> b -> b
```

Here dup is a function which copies its argument and returns the copies as a pair. The drop function is a function that ignores its first argument. Expressing drop as a function with type `a -> unit` is awkward since the “unit” may need to be dropped as well.

In Clamp, dup and drop are not defined as functions but are provided as primitives for syntactic convenience. To enforce substructural restrictions the instance rules for base types on these substructural type classes are also built in to the language.

With the addition of some new constraints on closure environments and the requisite instance rules, we can then build up a substructural type system encompassing U,R,A, and L types which consists of little more than a System-F core with linear types and standard type class constraints. There is no need for the of-course “!” operator, for qualifiers, or for kind systems. This is useful from a metatheoretic perspective, and also for programmers looking for a familiar frame in which to use substructural types or for a way to incorporate substructural types into existing compilers.

Compared with “!” based systems, using type classes to distinguish substructural types resolves the need for annotating types with “!” and supports much more polymorphism since type equality can be made independent of substructural usage. Type classes also alleviate the need for qualifier polymorphism in qualifier based systems, and also the need for subkinding polymorphism in kind based systems. For instance, the type for a pair constructor function in Clamp is simply

$$\forall \alpha, \beta\, [\,]\, .\alpha \xrightarrow{\mathbf{U}} \beta \xrightarrow{\mathbf{L}} \alpha \times \beta$$

where the substructural properties of the pair type $\alpha \times \beta$ are derived from the instance rules:

$$\text{instance Dup } \alpha, \text{Dup } \beta \implies \text{Dup } \alpha \times \beta$$

$$\text{instance Drop } \alpha, \text{Drop } \beta \implies \text{Drop } \alpha \times \beta$$

These tell us that a pair is duplicable when its components are, and similarly for drop.

Kind-based systems such as Alms in fact propagate constraints internally that are very similar to type class constraints, but type classes provide a more general framework, and structures such as the subkinding lattice are handled nicely by the implicit lattice structure of lists of constraints.

Furthermore, using separate type classes for the different substructural operations allows our system to extend orthogonally to all substructural types. In the context of reference cells, this means that many operations on refs (such as sharing pointers) that are forbidden in languages with only linear/affine types such as Alms or F° are allowable on the right types in Clamp.

Some systems such as λ^{URAL} also support the simultaneous usage of U, R, A, and L types, but they often require the introduction of a delicate system for managing contexts whose variable-typing bindings have different substructural properties. In Clamp, with the introduction of dup and drop operations, all variable bindings can be made strictly linear.

Finally, the explicit treatment of substructural dup and drop operations in our language leads to natural ways to incorporate memory control such as reference counting and strong reference updates.

Chapter 2

The Clamp Programming Language

2.1 Basic Features

The Clamp programming language is an ML-like language with a type class system that supports programming with substructural types. Unlike the λ_{cl} core calculus described in chapter 4, Clamp is a user-facing language. Thus, it contains some bells and whistles that make practical programming easier, but its type system is fundamentally based on λ_{cl}. The most significant difference between the two is that Clamp supports type inference, which allows its type checker to infer type schemes for the examples in this chapter without the need for type annotations.

The syntax and type system presented in this chapter correspond to the type checker I implemented (see Chapter 5). In particular, the syntax for Clamp is based off of OCaml, with inspiration for the substructural arrow syntax taken from Alms [Tov and Pucella, 2011]. For the sake of simplicity Clamp does not include named algebraic data-types or a module system, but there should be no specific difficulties in adding these features.

The first few subsections of this chapter will present Clamp in its more verbose syntax, where duplication (dup) and dropping (drop) are given explicitly. Later on, the dup and drop operations will be left implicit since they can be inferred by a separate pass of the type checker (described in chapter 3), but for the sake of clarity I will start by explicitly specifying all dup and drop operations.

2.1.1 Linearity

When annotated explicitly with dup and drop, Clamp at its core is a language with a syntactically linear type system: every variable once brought into scope must be used exactly once. The additional substructural types are all mediated by the dup and drop operations. Programming without using dup and drop operations is restrictive, but it is still possible to write functions which conserve their arguments, for instance a function that swaps the second components of two pairs. An implementation of such a function in Clamp is given in figure 2.1 on the next page

To take into account the fact that functions close over their environments, Clamp provides four different kinds of arrows denoted `-X>` for X corresponding to U(nlimited), A(ffine), R(elevant), or L(inear) functions. These functions support different substructural operations

```
fun p1 -L> fun p2 -L>
  letp (p1a, p1b) = p1 in
  letp (p2a, p2b) = p2 in
  ((p1a,p2b),(p2a,p1b))
```

Figure 2.1: Pair Component Swap

```
fun x -L> (x,x)    // BAD: reuses argument

fun x -L> 5        // BAD: ignores argument

fun x -L> fun y -L>
   match x with
    inl a -> (a, y)
  | inr b -> (b, b) // BAD: ignores y, reuses b
```

Figure 2.2: Nonlinear Functions

but impose corresponding constraints on their closure environments, and thus specify their substructural properties explicitly. The `-L>` used above creates linear functions which must be used exactly once, but impose no constraints on their environments.

In addition, to allow the usage of linear pairs, accessing the components of a pair in Clamp is done through the **letp** $(x_1, x_2) = e$ form which binds both components of a pair at once. Using the standard fst and snd operations would make it impossible to retrieve both components of a linear pair without using the pair twice.

Sums in Clamp do not require a special elimination form like pairs do, and one can use standard ML-style pattern matching to distinguish left and right injections.

As examples of the limitations enforced by a linear type system in the absence of dup and drop, the functions given in figure 2.2 are all ill-typed since they either reuse or ignore one of their arguments along one of their execution branches.

2.1.2 Dup and Drop

The dup and drop primitives in Clamp allow one to duplicate and dispose of values and variables, making explicit the substructural operations of contraction and weakening so that we can support U (unlimited: dup+drop), R (relevant: dup), and A (affine: drop) types. Operationally, one can think of dup and drop as copy and destruct operations on their arguments, though they can also be implemented with techniques such as reference counting.

For instance, the three ill-typed functions in figure 2.2 can be rewritten in figure 2.3 on the facing page using dup and drop to satisfy linear usage restrictions. In figure 2.3, each variable is used exactly once along every execution branch. Note that the dup primitive has a slightly complicated syntax in Clamp since it must rebind its arguments with fresh names to allow reuse and satisfy linearity.

Dup and drop provide a way to make all usage in Clamp appear syntactically linear

```
fun x -L>
  dup x as (x0,x1) in
  (x0,x1)

fun x -L> drop x in 5

fun x -L> fun y -L> match x with
        inl a -> (a,y)
      | inr b ->
          drop y in
          dup b as (b0,b1) in
          (b0,b1)
```

Figure 2.3: Nonlinear Functions with dup/drop

since they mediate all nonlinear usage of values and variables. In Clamp then, the task of ensuring that values (with different substructural properties) are used correctly is reduced to enforcing constraints on the arguments to dup and drop.

Suppose we have a type system where file descriptors are linear such as in Chapter 1. If *fd* refers to a file descriptor then the expression `drop fd in 4` is ill-typed because drop cannot be applied to a value of type "file".

As another example, consider a metaphorical treatment of gold coins of type "gold". A function minegold : unit $\xrightarrow{\mathbf{U}}$ gold for mining gold coins might be provided. In this system, units of gold can be lost, but gold cannot be duplicated. Thus "gold" would be a relevant type, which can be dropped but not duplicated.

In the framework of type classes, one can call `drop` on a value with type τ if there exists an instance of the predicate Drop τ, and in this case there is no instance for Drop file. The dup and drop operations are the sole type class methods of the Dup and Drop type classes, and introduce the corresponding constraints.

Going back to the code examples in figure 2.3, the first function might be assigned the type int $\xrightarrow{\mathbf{L}}$ int $\times$ int since integers are duplicable, but not the type gold $\xrightarrow{\mathbf{L}}$ gold $\times$ gold because there would be no instance for Dup gold. It is also possible to assign the first function a constrained polymorphic type, which will be discussed later in the chapter.

Dup and drop provide explicit ways of managing substructural operations while keeping the core language consistently linear. However, since they are cumbersome for the user to write, in chapter 3 we give an optimal algorithm for inferring these operations automatically. Given a program which uses variables arbitrarily, this algorithm examines where variables are used and inserts a minimal number of dup and drop operations, renaming variable usages as it goes, so that the annotated program appears syntactically linear. The task of examining whether dup and drop operations were used on variables with legal types can then be left to the main type checker. Given the functions in figure 2.2, a dup/drop insertion pass in fact infers the annotations given in figure 2.3 and then the main type inference routine can be run on the latter.

Throughout the rest of the chapter we will leave the dup and drop operations implicit, assuming that our type checker will run after an inference algorithm has inserted them and renamed variables wherever they are needed.

2.1.3 Substructural Constraints and Type Class Instances

Base types in Clamp such as unit and int support both dup and drop (e.g. there are instances of Dup τ and Drop τ for τ = unit, int). The arrow types also explicitly specify which operations they support since for instance $a \xrightarrow{\mathbf{R}} b$ supports dup but not drop. However, in Clamp as in kind-based systems such as F° and *Alms*, compound forms such as pairs and sums do not specify their substructural properties explicitly. These are instead determined by instance rules. For example, for pairs we have the type class instance rule:

$$\text{instance Dup } a, \text{Dup } b \implies \text{Dup } (a \times b)$$

which allows us to derive that $(1, 1)$ is a duplicable pair since integers are duplicable. Since there are no other Dup instance rules for pairs, the code below is invalid because we cannot derive Dup (int × gold)

```
// @minegold : unit -U> gold

let mygold = @minegold unit in
(fun a -L> (a,a)) (1,mygold) //BAD: no Dup instance for gold
```

Note that the "@" syntax above is used to prefix global identifiers which are not built-in primitives in Clamp. The dup/drop insertion pass can then ignore them since global identifiers are required to be both duplicable and droppable.

Sound substructural instance rules for all of the built-in types in Clamp are built-in to the type checker. Clamp does not yet provide a mechanism for defining user instances, but it would be straightforward to extend the language to allow for user defined type classes and instances as in Haskell, and to give a way for programmers to specify new types with varying substructural instance rules.

Allowing users to define custom implementations (as opposed to just static instance rules) for the dup and drop methods would be more difficult to incorporate into the language. This issue is discussed in the future work section (section 6.2).

2.1.4 References

References have very delicate instance rules for their substructural properties. Clamp has both weak and strong references with types denoted by $\text{ref}^{\text{rq}}\tau$ with a qualifier rq.

$$\text{rq} ::= \text{s (strong), w (weak)}$$

To access the data inside a reference, Clamp uses a swap primitive, which take in a reference cell and a value, and returns both an updated reference cell and the old value stored inside.

	Type	Strong Ref Support	Weak Ref Support
swap	$\mathrm{ref}^{\mathrm{rq}}\alpha \times \alpha \xrightarrow{\mathbf{U}} \mathrm{ref}^{\mathrm{rq}}\alpha \times \alpha$	x	x
sswap	$\mathrm{ref}^{\mathrm{s}}\alpha \times \beta \xrightarrow{\mathbf{U}} \mathrm{ref}^{\mathrm{s}}\beta \times \alpha$	x	
release	$\mathrm{ref}^{\mathrm{rq}}\alpha \xrightarrow{\mathbf{U}} \mathrm{unit} + \alpha$	x	x
srelease	$\mathrm{ref}^{\mathrm{s}}\alpha \xrightarrow{\mathbf{U}} \alpha$	x	

Table 2.1: Reference Types in Clamp

Without a swap operation, there would be no way to read from a reference without implicitly copying its contents, and vice versa for writes. In table 2.1 we give the operations supported by both strong and weak references.

Note that the swap and sswap operations allow for weak and strong updates on the appropriate references, and there are also two separate "release" operations for deallocating a reference. These are provided because it would be illegal to directly drop references that contain relevant or linear contents. In a strong reference, releasing a reference deallocates it and returns its contents so that the contents are neither copied nor lost. For weak references (which can be aliased) we adopt the convention of giving release the option to return either unit or the cell contents depending on the number of remaining live links into the reference.

We would like to be able to alias weak references so as to share them, but aliasing strong references allows for unsound usage when two segments of code assume that a strong reference contains different types. In the code below, we perform a strong update on a cell containing a function. However, we then try to perform another strong update, expecting the reference r to still contain a function when it now contains an integer.

```
fun r -L>
  (letp (r1,r1val) = sswap (r,1) in (r1val 1),
   letp (r2,r2val) = sswap (r,2) in (r2val 2))
```

Thus in Clamp, there is no instance rule for duplicating a strong reference, and there are other restrictions which will be described in chapter 4.

2.1.5 Restrictions on Arrows

The Dup and Drop instances for the different arrow types are simple because they depend directly on the arrow qualifier. However, we must be careful in tracking the variables captured by a function's closure. At a lower level, a function type is an abstraction over an environment paired with code, so the substructural properties of a function must depend on that of its environment.

In fact, different forms of arrows impose different Dup and Drop constraints on their closure environments to ensure the consistency of their own use. For instance, in figure 2.4 on the following page we present two partial applications of curried pair functions.

The first partial application "partial1" yields a linear function (`-L>`), which cannot be duplicated or dropped, so it is valid for this function to close over the gold passed in. The

```
// Well-Typed
let mygold1 = @minegold unit in
let partial1 =
        (fun x -U> fun y -L> (x,y)) mygold1

// Ill-Typed: gold hidden inside unlimited closure
let mygold2 = @minegold unit in
let partial2 =
        (fun x -U> fun y -U> (x,y)) mygold 2
```

Figure 2.4: Partial Application Closures

gold cannot be secretly duplicated by duplicating the linear function.

However, the second partial application "partial2" yields an unlimited function, which can be duplicated and dropped, so it is not well typed. A function which claims to support a substructural operation imposes the same constraint on its environment. In this case, an unlimited function can only be well typed if its closure environment satisfies both Dup and Drop constraints.

2.1.6 Polymorphism

As seen in Haskell, type class constraints work closely with support for polymorphism. If we revisit the original three nonlinear functions in figure 2.3 on page 15, we (and the implemented type checker) can in fact infer the constrained type schemes below. In this notation, $\forall\overline{\alpha_i}\,[P]\,.\tau$ denotes a type scheme where the type variables $\overline{\alpha_i}$ can be instantiated with any τ_i so long as they satisfy the constraints in P.

$$\forall\alpha\,[\text{Dup }\alpha]\,.\alpha \xrightarrow{\mathbf{L}} \alpha\times\alpha$$

$$\forall\alpha\,[\text{Drop }\alpha]\,.\alpha \xrightarrow{\mathbf{L}} \text{int}$$

$$\forall\alpha\,[\text{Dup }\alpha, \text{Drop }\alpha]\,.\alpha+\alpha \xrightarrow{\mathbf{L}} \alpha \xrightarrow{\mathbf{L}} \alpha\times\alpha$$

In Haskell-like syntax, these would be:

```
Dup a => a -L> (a,a)
Drop a => a -L> Int
Dup a, Drop a => Either a a -L> a -L> (a,a)
```

When these type schemes are instantiated with a type, the type system will check that the type supports the appropriate constraints specified in the type scheme. In figure 2.5 on the facing page, it is possible to infer a polymorphic type scheme for the function *f*, but the later application is ill-typed since the argument does not satisfy the constraint in the type scheme for *f*.

```
let f = fun x -L>
  dup x as (x0,x1) in
  (x0,x1)
  // f : Dup a => a -L> (a,a)
  in
  (f (@minegold unit)) // BAD: no Dup instance for gold type
```

Figure 2.5: Bad instantiation

```
// some : a -U> sum a b
let some = fun x -U> inl x in
  (some 1,some @lunit)

// fst : Drop b => pair a b -U> a
let fst = fun p -U>
  letp (p1, p2) = p in
  p1

// mappair : Dup x => (a -x> b) -U> pair a a -L> pair b b
let mappair = fun f -U> fun p -L>
  letp (p1,p2) = p in
  (f p1, f p2)
```

Figure 2.6: Polymorphism Examples

Type schemes with type class constraints provide much of the same bounded polymorphism that kind based substructural systems use subkinding to achieve, and which is furthermore awkward in many substructural and linear type systems. In figure 2.6 for instance, the "some" function has an inferred polymorphic type which allows it to be applied to any argument, regardless of its substructural properties. The "fst" function also has an inferred type scheme which allows it to operate on any pair so long as the second component has a droppable type. The pair itself is unrestricted in any other way, and itself could either be droppable or not.

One weakness of the Clamp type system is that arrows are assigned a concrete substructural qualifier. For instance, in writing a curried pair function, one might want to specify that a partial application `pair x` is duplicable when "x" is duplicable. However, in Clamp, one must decide the substructural properties of a function up front, and assign it a U, R, A, or L qualifier.

Despite this, type classes are a flexible enough framework that we can emulate some degree of polymorphism by making the $\to$ type constructor take 3 type arguments, in other words to give it the kind $\star \to \star \to \star \to \star$. The first argument will be a dummy type U, R, A, or L which specifies the substructural properties of the arrow, and the second and third arguments will be the domain and range of the arrow. With this usage of dummy types,

types such as int $\xrightarrow{\text{int}}$ int are well-formed but uninhabited, since every concrete function is created with a specified qualifier U, R, A, or L.

By using dummy types as qualifiers in this way, type classes allow us a form of bounded polymorphism on arrows as well. For instance, in the "mappair" example in figure 2.6 on the preceding page, the type scheme specifies that mappair can map any function which is duplicable, which includes both `-R>` and `-U>` arrows. This treatment of arrow qualifiers is implemented in the type checker, but is not modeled in the core calculus given in Chapter 4. A similar extension allows the type checker to infer general type schemes on code which uses weak reference operations. For instance, the function `fun r -U> swap (r,1)` can be given the type $\forall\alpha\,[]\,.\text{ref}^{\text{rq}}\ \text{int} \xrightarrow{\mathbf{U}} (\text{ref}^{\text{rq}}\ \text{int}) \times \text{int}$, allowing it to be applied to both strong and weak references.

2.2 Further Examples

As described in Chapter 1, the managing of resources with precise state is one of the strengths of using substructural types.

2.2.1 File Handles

If Clamp had support for modules, one could imagine exporting a file I/O library with the following signature:

```
type fhandle // no Dup/Drop instances for fhandle!

val open : unit -U> fhandle
val close : fhandle -U> unit
val read : fhandle -U> pair int fhandle
```

However, though there is nothing in Clamp that prevents the addition of modules, they are not implemented and for now we can provide a dummy implementation which uses a linear unit type to simulate a file handle. The @lunit builtin identifier creates a linear unit, and the @ldispose operation consumes a linear unit and disposes of it to return a regular unit.

```
let open = fun u -U> @lunit u in
let close = fun f -U> @ldispose f in
let read = fun f -U> (1,f) in    // always reads in a "1"
```

Then, given the above operations, we can write functions which manipulate files.

The function "goodmanip" defined in figure 2.7 on the next page is straightforward, but the power of substructural types comes from the fact that using the file operations in any sequence but the correct one would be a type error. For instance, the function "badmanip" attempts to read from a file after it has been closed. It would not type-check for two reasons.

```
let goodmanip = fun u -U>
  let f = open u in
  letp (data1,f1) = read f in
  let nil = close f1 in
  data1

let badmanip = fun u -U>
  let f = open u in
  let nil = close f1 in
  letp (data1,f2) = read f in
  data1
```

Figure 2.7: File Manipulation

Firstly, the file handle *f* was duplicated even though file handles are linear, and secondly the file handle *f2* was ignored and not disposed of properly.

2.2.2 Shared Pointers

The file example above makes use of linear types, but Clamp supports much more of the substructural lattice. The combination of U,R,A, and L types as well as strong and weak references allows for very fine grained control of resources in a program.

As another example, consider a situation where one would like to run two threads concurrently given a "runParallel" function. One would like both of the threads to have access to a common file handle for error logging, but one would also like to ensure that the file is properly closed exactly once in the end.

One way to do this would be to define a single shared weak reference which contains a file handle. File handles cannot be duplicated, but weak references can be, so both threads can have access to a common file in this way. In addition, the presence of relevant types in Clamp makes it possible to guarantee that each spawned thread will attempt to close its file, while the reference counted dynamics of weak references ensure that both the reference cell and the file are deallocated properly in the end. The code in figure 2.8 on the following page is an example of how this could be implemented in Clamp, assuming the file manipulation library described earlier and a dummy implementation of runParallel.

```
let runParallel = fun f -U> fun x -L>
  (f x, f x) in

let fref = wnew (open unit) in
let proc = (fun u -R>
   match (release fref) with
    inl f -> close f
  | inr dummy -> dummy
  ) in

runParallel proc unit
```

Figure 2.8: Shared File Handles

Chapter 3

Inferring Substructural Operations

As we saw earlier, tracking substructural operations in terms of explicit dup and drop operations is both flexible and natural. However, most conventional programming still relies heavily on implicit copying and discarding, and inserting these operations by hand can be a frustrating experience. For instance, consider the function to (approximately) solve the quadratic formula in figure 3.1.

```
// findroot : int -U> int -U> int -U> sum int unit
let findroot = fun a -U> fun b -U> fun c -U>
  let discr = b*b-4*a*c in
  if (discr > 0) then
    inl (-b+@sqrt(discr))/(2*a)
  else
    inr ()
```

Figure 3.1: Quadratic Formula Example

To make this function type check, we would need to annotate it as in figure 3.2 on the following page so that every variable is used once along every execution path.

To address this issue, in this chapter we develop an algorithm for inserting dup and drop operations automatically into unannotated code. Since different annotations can lead to different static and dynamic semantics, we prove that our algorithm generates an *optimal* annotation in two senses:

- It minimizes the program's memory usage (Theorem 3.2)
- It imposes the minimal set of type class constraints (Theorem 3.3)

Other languages which make use of explicit substructural operations such as linear lisp [Baker, 1994] may find the algorithms given in this chapter generally applicable as well. The "infer" algorithm developed in this chapter shares many similarities with the linear use-type reconstruction algorithm in Wadler [1991], but none of the relevant optimality properties are established for that algorithm.

```
// findroot : int -U> int -U> int -U> sum int unit
let findroot = fun a -U> fun b -U> fun c -U>
  dup a,b as (a1,a2),(b1,b2) in
  let discr =
    dup b1 as (b3,b4) in
    b3*b4-4*a1*c in
  dup discr as (discr1,discr2) in
  if (discr1 > 0) then
    inl (-b2+@sqrt(discr2))/(2*a2)
  else
    drop a2,b2,discr2 in
      inr unit
```

Figure 3.2: Quadratic Formula with dup/drop

3.1 A Core Linear Language

To focus on the essential problems, we will start by examining an abstraction of the linear lambda calculus, λ_{lin}

λ_{lin} **syntax**

$$e ::= x \mid \lambda x.e \mid \langle e_1, e_2 \rangle \mid [e_1, e_2]$$

$$ae ::= x \mid \lambda x.ae \mid \langle ae_1, ae_2 \rangle \mid [ae_1, ae_2] \mid \textbf{dup } \Gamma \textbf{ in } ae \mid \textbf{drop } \Gamma \textbf{ in } ae$$

This untyped calculus is designed to model just the substructural parts of a type system (those that track variable usage counts), allowing us to focus on inserting dup and drop operators independently of the underlying base types (int, string, etc.). The $\langle e_1, e_2 \rangle$ form constructs a pair, and the $[e_1, e_2]$ is the linear-logic “with” or “&” form, a sort of delayed branching computation where one of the branches is later chosen for evaluation. Terms e which are *expressions* (abbreviated *exp*) are unannotated and don’t explicitly satisfy substructural usage constraints. Terms ae which are *annotated expressions* (abbreviated *aexp*) are annotated with dup and drop operations to allow for explicit nonlinear usage of variables. The dup and drop operations work over contexts Γ which are multisets of variables, and which will be described in the next section.

We assume implicitly that all variables in λ_{lin} have distinct names to avoid dealing with alpha-conversion and shadowing. This is straightforward to achieve in a real language with the addition of a renaming pass

Linear Multiset Environments

$$\Gamma \quad : \quad \text{var} \to \text{nat}$$

$$\Gamma_1 + \Gamma_1 = \lambda(x).\Gamma_1(x) + \Gamma_2(x) \qquad \Gamma_1 \sqsubseteq \Gamma_2 \equiv \forall x.\Gamma_1(x) \leq \Gamma_2(x)$$

$$\Gamma_1 - \Gamma_1 = \lambda(x).\Gamma_1(x) - \Gamma_2(x) \qquad x \in \Gamma \equiv \Gamma(x) > 0$$

$$\Gamma_1 \sqcup \Gamma_1 = \lambda(x).\max\ (\Gamma_1(x), \Gamma_2(x)) \qquad \{x\} \equiv \lambda(y).\begin{cases} 1 & y = x \\ 0 & y \neq x \end{cases}$$

$$\Gamma_1 \sqcap \Gamma_1 = \lambda(x).\min\ (\Gamma_1(x), \Gamma_2(x))$$

Figure 3.3: Environment Operations

3.1.1 Linear Environments

The environments or contexts (abbreviated *ctxt*) Γ in this system only manage scope and binding by restricting contraction and weakening to explicit dup and drop annotations. They do not track actual types. In order to avoid for now the messy but straightforward process of generating names and renaming variables when inserting a dup, we think of contexts Γ as *multisets* $\{x, x, y, z, \ldots\}$ of in-scope variables, defined mathematically as functions from variables to counts. We can then define multiset operations on contexts in figure 3.3.

These operations include combining contexts to join them either by lifting the arithmetic $+$ and $-$ operators or by extending the concept of set union and set intersection into $\sqcup$ and $\sqcap$ operators. One important detail in our definitions is that we interpret $\Gamma : \mathit{var} \to \mathit{nat}$ for the *naturals: 0,1,2,...* Thus $(x : \mathit{nat}) - (y : \mathit{nat}) = 0$ when $y \geq x$.

Then, we can use these linear multiset contexts to define a notion of *well-formedness* in figure 3.4 on the following page, which describes when an annotated term *ae* in λ_{lin} accounts for all nonlinear usage of its variables through explicit dup and drop operations.

$$\boxed{\text{Linear Well Formedness: } \Gamma \vdash ae}$$

The L-Var, L-Abs, L-Pair, and L-Choice rules all follow standard linear lambda calculi. The L-Var rule requires its context to contain exactly the variable in question so that no variables are used more or less than the context allows. The environment is split in the L-Pair rule since a usage of a variable adds towards the total usage count in either component of a pair, but the environment in the L-Choice rule is shared since no more than one branch of a "with" form can be taken. The L-Dup and L-Drop rules allow one to perform contraction and weakening on a context as desired, so long as they are specified explicitly in the **dup** or **drop** forms.

Linearity Rules

$$\frac{}{\{x\} \vdash x}\ \textsc{L-Var} \qquad \frac{\Gamma + \{x\} \vdash ae \qquad x \notin \Gamma}{\Gamma \vdash \lambda x.ae}\ \textsc{L-Abs} \qquad \frac{\Gamma_1 \vdash ae_1 \qquad \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2 \rangle}\ \textsc{L-Pair} \qquad \frac{\Gamma \vdash ae_1 \qquad \Gamma \vdash ae_2}{\Gamma \vdash [ae_1, ae_2]}\ \textsc{L-Choice}$$

$$\frac{\Gamma_1 + \Gamma_2 + \Gamma_2 \vdash ae}{\Gamma_1 + \Gamma_2 \vdash \textbf{dup}\ \Gamma_2\ \textbf{in}\ ae}\ \textsc{L-Dup} \qquad \frac{\Gamma_1 \vdash ae}{\Gamma_1 + \Gamma_2 \vdash \textbf{drop}\ \Gamma_2\ \textbf{in}\ ae}\ \textsc{L-Drop}$$

Figure 3.4: Well Formedness Rules

3.1.2 Encoding a Full Language

Many of the forms in Clamp are not included in λ_{lin}, for instance function application $e_1\ e_2$ or the **match** construct. This is because we can encode the usage behavior (binding structure) of every other construct in λ_{cl} in terms of the basic ones in λ_{lin}. For instance, we can extend our inference algorithm to map over applications $e_1\ e_2$ exactly the same way we map over pairs (e_1, e_2) since the environment splitting behavior is the same. In figure 3.5 on the next page we present reductions for a selection of the syntactic forms in λ_{cl} which will be introduced in chapter 4. e^* is used to denote the homomorphic mapping of e under the reduction.

As an example, intuitively we can encode a **letp** $(x_1, x_2) = e$ **in** e_2 form in terms of pairs and lambdas because the variable usage behavior of **letp** is merely to add up the usage behavior of e and e_2 just as a pair would, except to bind two new variables x_1, x_2 in e_2 just like two lambdas would.

3.2 An Annotation Inference Algorithm

Given an unannotated core term e, we would like our inference algorithm to return an annotated ae which is well formed. To propagate variable usage information up the AST, it turns out to be easiest to have the function return both an annotated ae and an environment Γ such that $\Gamma \vdash ae$. Our inference function infer$_s$ thus has type $exp \to aexp \times ctxt$, and is defined in figure 3.6 on the facing page.

The intuition behind this algorithm is to try and minimize the context required to validate each subterm by propagating variable usages up the AST. This formulation for infer$_s$ is comparatively easy to specify and especially suitable for implementation. However, to simplify the proofs in this chapter it is convenient to have the inference function return not just a term but a *well-formedness derivation.* Thus we extend infer$_s$ to a more explicit infer$_d$ with type $exp \to \Gamma \vdash ae$ in figure 3.7 on page 28. For the rest of the chapter unless otherwise stated the "inference algorithm" refers to infer$_d$.

Syntactic Reductions

$$x \leadsto x \qquad \lambda^{aq}(x:\tau).e \leadsto \lambda x.e^*$$

$$e_1\ e_2 \leadsto \langle e_1^*, e_2^* \rangle \qquad \textbf{inl}\ e, \textbf{inr}\ e \leadsto e^*$$

$$\textbf{new}^{\textbf{rq}}\ e \leadsto e^* \qquad \textbf{release}^{\textbf{rq}}\ e \leadsto e^*$$

$$\textbf{swap}^{\textbf{rq}}\ e_1\ \textbf{with}\ e_2 \leadsto \langle e_1^*, e_2^* \rangle$$

$$\textbf{letp}\ (x_1, x_2) = e_1\ \textbf{in}\ e_2 \leadsto \langle e_1^*, \lambda x_1.\lambda x_2.e_2^* \rangle$$

$$\textbf{match}\ e\ \textbf{with inl}\ x_1 \to e_1; \textbf{inr}\ x_2 \to e_2 \leadsto \langle e^*, [\lambda x_1.e_1^*, \lambda x_2.e_2^*] \rangle$$

$$\dots$$

Figure 3.5: Reducing λ_{cl} to λ_{lin}

Inference Algorithm

$$\text{infer}_s \quad : \quad exp \to aexp \times ctxt$$

$$\text{infer}_s(x) = (x, \{x\})$$

$$\text{infer}_s(\lambda x.e) = \text{let}\ (ae_1, \Gamma_1) = \text{infer}_s(e)$$

$$\begin{cases} (\lambda x.ae_1, \Gamma_1 - \{x\}) & \text{when } x \in \Gamma_1 \\ (\lambda x.\textbf{drop}\ x\ \textbf{in}\ ae_1, \Gamma_1) & \text{when } x \notin \Gamma_1 \end{cases}$$

$$\text{infer}_s(\langle e_1, e_2 \rangle) = \text{let}\ (ae_i, \Gamma_i) = \text{infer}_s(e_i)$$

$$(\textbf{dup}\ \Gamma_1 \sqcap \Gamma_2\ \textbf{in}\ \langle ae_1, ae_2 \rangle, \Gamma_1 \sqcup \Gamma_2)$$

$$\text{infer}_s([e_1, e_2]) = \text{let}\ (ae_i, \Gamma_i) = \text{infer}_s(e_i)$$

$$([\textbf{drop}\ \Gamma_2 - \Gamma_1\ \textbf{in}\ ae_1, \textbf{drop}\ \Gamma_1 - \Gamma_2\ \textbf{in}\ ae_2], \Gamma_1 \sqcup \Gamma_2)$$

Figure 3.6: Inference Algorithm

Derivation Inference

$$\text{infer}_d \quad : \quad exp \to \Gamma \vdash ae$$

$$\text{infer}_d\,(x) = \frac{}{x \vdash x}$$

$$\text{infer}_d\,(\lambda x.e) = \text{let } \mathcal{D} :: \Gamma_1 \vdash ae_1 = \text{infer}_d\,(e)$$

$$\begin{cases} \dfrac{\mathcal{D} = \dfrac{\cdots}{\Gamma_1 \vdash ae_1}}{\Gamma_1 - \{x\} \vdash \lambda x.ae_1} & \text{when } x \in \Gamma_1 \text{so } \Gamma_1 = \Gamma_{11} + \{x\} \\ \dfrac{\dfrac{\mathcal{D} = \dfrac{\cdots}{\Gamma_1 \vdash ae_1}}{\Gamma_1, x \vdash \textbf{drop } x \textbf{ in } ae_1}}{\Gamma_1 \vdash \lambda x.\textbf{drop } x \textbf{ in } ae_1} & \text{when } x \notin \Gamma_1 \end{cases}$$

$$\text{infer}_d\,(\langle e_1, e_2\rangle) = \text{let } \mathcal{D}_i :: \Gamma_i \vdash ae_i = \text{infer}_d\,(e_i)$$

$$\dfrac{\dfrac{\mathcal{D}_1 = \dfrac{\cdots}{\Gamma_1 \vdash ae_1} \qquad \mathcal{D}_2 = \dfrac{\cdots}{\Gamma_2 \vdash ae_2}}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2\rangle}}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{dup } \Gamma_1 \sqcap \Gamma_2 \textbf{ in } \langle ae_1, ae_2\rangle}$$

$$\text{infer}_d\,([e_1, e_2]) = \text{let } \mathcal{D}_i :: \Gamma_i \vdash ae_i = \text{infer}_d\,(e_i)$$

$$\dfrac{\dfrac{\mathcal{D}_1 = \dfrac{\cdots}{\Gamma_1 \vdash ae_1}}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{drop } \Gamma_2 - \Gamma_1 \textbf{ in } ae_1} \qquad \dfrac{\mathcal{D}_2 = \dfrac{\cdots}{\Gamma_2 \vdash ae_2}}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{drop } \Gamma_1 - \Gamma_2 \textbf{ in } ae_2}}{\Gamma_1 \sqcup \Gamma_2 \vdash [\textbf{drop } \Gamma_2 - \Gamma_1 \textbf{ in } ae_1, \textbf{drop } \Gamma_1 - \Gamma_2 \textbf{ in } ae_2]}$$

Figure 3.7: Inference Algorithm on Derivations

3.3 Algorithm Soundness and Optimality

3.3.1 Preliminaries

Throughout the proofs below, we will need the multiset properties in Lemma 3.1 to verify that the inference algorithm is sound.

Lemma 3.1 (Multiset Lemmas 1). *Multiset Properties:*

- $(\Gamma_1 \sqcup \Gamma_2) + (\Gamma_1 \sqcap \Gamma_2) = \Gamma_1 + \Gamma_2$
- $(\Gamma_1 - \Gamma_2) + \Gamma_2 = \Gamma_1 \sqcup \Gamma_2$

Proof. See appendix A, page 63 □

Additionally, note that we defined the operations of $+, -, \sqcup, \sqcap, \sqsubseteq$ by lifting the operations of $+$, $-$, max, min, and $\leq$ defined on naturals onto functions. Thus we can verify without proof that various other properties involving natural numbers also hold in their multiset liftings.

Lemma 3.2 (Multiset Lemmas 2). *Lifting Lemmas:*

- *If* $\Gamma_1 \sqsubseteq \Gamma_1'$ *and* $\Gamma_2 \sqsubseteq \Gamma_2'$ *then* $\Gamma_1 + \Gamma_2 \sqsubseteq \Gamma_1' + \Gamma_2'$
- *If* $\Gamma_1 \sqsubseteq \Gamma_2$ *then* $\Gamma_1 - \Gamma_3 \sqsubseteq \Gamma_2 - \Gamma_3$ *for any* Γ_3
- $\Gamma_1 + \Gamma_2 - \Gamma_2 = \Gamma_1$

We also define the function erase (ae) which relates annotated terms to their core unannotated forms.

Erase

$$\text{erase} \quad : \quad aexp \to exp$$

$$\text{erase}(x) = x \qquad \text{erase}(\lambda x.e) = \lambda x.\text{erase}(e)$$

$$\text{erase}(\langle e_1, e_2\rangle) = \langle \text{erase}(e_1), \text{erase}(e_2)\rangle \qquad \text{erase}([e_1, e_2]) = [\text{erase}(e_1), \text{erase}(e_2)]$$

$$\text{erase}(\textbf{dup}\ \Gamma\ \textbf{in}\ e) = \text{erase}(e) \qquad \text{erase}(\textbf{drop}\ \Gamma\ \textbf{in}\ e) = \text{erase}(e)$$

With this we can proceed to show that the inference algorithm is sound.

Theorem 3.1 (Inference Soundness). *For any e,* $\text{infer}_d(e)$ *is a valid derivation of* $\Gamma \vdash ae$ *for some* Γ *and* ae *where* $\text{erase}(ae) = e$.

Proof. By induction on e, making use of Lemma 3.1. See appendix A, page 64 for details. □

Then, since most implementations will want to use infer_s, we can derive the corollary that the simpler inference algorithm is sound as well.

Corollary 3.1. *If* $\text{infer}_s(e) = (ae, \Gamma)$ *then* $\Gamma \vdash ae$ *and* $\text{erase}(ae) = e$

Proof. By inspection, we can verify that $\text{infer}_s(e) = (ae, \Gamma)$ iff $\text{infer}_d(e) :: \Gamma \vdash ae$. □

3.3.2 Memory Usage

Though we have not given a semantics for the memory usage of programs yet, if we have a valid derivation $\Gamma \vdash ae$ intuitively it means that ae requires requires all of the variables in Γ to be in-memory and live to execute. Thus, we aim to prove that these "live-sets" (environments) Γ are minimized by our algorithm throughout a program. These results will also be essential in proving that the inference algorithm duplicates and drops only when necessary.

We introduce some functions which will prove useful in identifying the *minimal* environments required by an annotated term. First the function fv returns the standard set of free variables in a term, and fv′ is a closely related function defined in terms of fv which is just fv in almost all places except where a variable is used by both components of a pair. Note that $\text{fv}\,(e)$ returns a set but we will often implicitly treat it as a multiset that satisfies the set invariants.

Free Variables

$$\begin{array}{rcl} \text{fv} & : & \mathit{exp} \to \mathit{set}\,(\mathit{var}) \\ \text{fv}\,(x) = \{x\} \\ \text{fv}\,(\lambda x.e) = \text{fv}\,(e) \setminus \{x\} \\ \text{fv}\,(\langle e_1, e_2\rangle) = \text{fv}\,(e_1) \cup \text{fv}\,(e_2) \\ \text{fv}\,([e_1, e_2]) = \text{fv}\,(e_1) \cup \text{fv}\,(e_2) \end{array} \qquad \begin{array}{rcl} \text{fv}' & : & \mathit{exp} \to \mathit{lenv} \\ \text{fv}'\,(\langle e_1, e_2\rangle) = \text{fv}\,(e_1) + \text{fv}\,(e_2) \\ \text{fv}'\,(e) = \text{fv}\,(e) \qquad \text{otherwise} \end{array}$$

With these we can establish lower bounds on the sizes of the environments of annotated expressions. The next few lemmas show that fv is a function which tells us the minimum environment an expression needs so that a valid annotated expression can be inferred for it, while fv′ tells us the minimum environment needed to infer an annotated expression that doesn't change the root AST constructor of an expression (e.g. doesn't wrap an expression in a dup/drop at the root).

Lemma 3.3 (Environments include Free Variables)**.** *If* $\Gamma \vdash ae$ *then* $\mathit{fv}\,(\text{erase}\,(ae)) \sqsubseteq \Gamma$

Proof. By induction on the derivation $\mathcal{D}$ for $\Gamma \vdash ae$, taking advantage of the fact that fv returns a Set, a collection which contains at most one copy of each element.

The induction hypothesis will tell us that for any derivation $\mathcal{D}$, any subderivation $\mathcal{D}_1 :: \Gamma_1 \vdash ae_1$ satisfies $\text{fv}\,(\text{erase}\,(ae_1)) \sqsubseteq \Gamma_1$.

Three representative cases are presented below:

Case L-Pair: $ae = \langle ae_1, ae_1\rangle$ and $\mathcal{D} = \dfrac{\Gamma_1 \vdash ae_1 \qquad \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2\rangle}$.

By the induction hypothesis, $\text{fv}\left(\text{erase}\left(ae_1\right)\right) \sqsubseteq \Gamma_1$ and $\text{fv}\left(\text{erase}\left(ae_2\right)\right) \sqsubseteq \Gamma_2$. Now,

$$\begin{aligned} \text{fv}\left(\text{erase}\left(\langle ae_1, ae_2\rangle\right)\right) &= \text{fv}\left(\langle \text{erase}\left(ae_1\right), \text{erase}\left(ae_2\right)\rangle\right) \\ &= \text{fv}\left(\text{erase}\left(ae_1\right)\right) \cup \text{fv}\left(\text{erase}\left(ae_2\right)\right) \\ &= \text{fv}\left(\text{erase}\left(ae_1\right)\right) \sqcup \text{fv}\left(\text{erase}\left(ae_2\right)\right) \end{aligned}$$

By Lemma 3.2, we can conclude that $\text{fv}\left(\text{erase}\left(\langle ae_1, ae_2\rangle\right)\right) \sqsubseteq \Gamma_1 + \Gamma_2$ as desired.

Case L-Abs: $ae = \lambda x.ae_1$ and $\mathcal{D} = \dfrac{\Gamma + \{x\} \vdash ae_1}{\Gamma \vdash \lambda x.ae_1}$.

By the induction hypothesis, $\text{fv}\left(\text{erase}\left(ae_1\right)\right) \sqsubseteq \Gamma + \{x\}$. Now by Lemma 3.2, $\text{fv}\left(\text{erase}\left(ae_1\right)\right) - \{x\} \sqsubseteq \Gamma + \{x\} - \{x\}$ so $\text{fv}\left(\text{erase}\left(\lambda x.ae_1\right)\right) \sqsubseteq \Gamma$ and we are done.

Case L-Dup $ae = \textbf{dup}\ \Gamma_2\ \textbf{in}\ ae_1$ and $\mathcal{D} = \dfrac{\Gamma_1 + \Gamma_2 + \Gamma_2 \vdash ae_1}{\Gamma_1 + \Gamma_2 \vdash \textbf{dup}\ \Gamma_2\ \textbf{in}\ ae_1}$.

By the induction hypothesis $\Gamma' = \text{fv}\left(\text{erase}\left(ae_1\right)\right) \sqsubseteq \Gamma_1 + \Gamma_2 + \Gamma_2$. Consider $y \in \Gamma'$. Since $\Gamma' \sqsubseteq \Gamma_1 + \Gamma_2 + \Gamma_2$, $y \in \Gamma_1$ or $y \in \Gamma_2$ or both. In either case, $y \in \Gamma_1 + \Gamma_2$. However, since Γ' is a set, it contains at most one copy of each variable, so $\Gamma' \sqsubseteq \Gamma_1 + \Gamma_2$.

Thus $\text{fv}\left(\text{erase}\left(ae\right)\right) \sqsubseteq \Gamma_1 + \Gamma_2$ □

In order to relate annotated and unannotated expressions, we need a way to identify the portions of annotated expressions that correspond to their unannotated components. This is especially important in quantifying the sizes of the variable context at various points, since we would like to ignore differences due to permuting or combining consecutive dup or drop operations. Instead it is easier to focus on the size of the environment at points in the AST which correspond to points in an unannotated expression. Thus we introduce the concept of bare expressions.

Definition 3.1. Let bare : $aexp \to bool$ be a predicate on annotated expressions describing if an expression is unannotated at its root level, i.e. if it is not rooted at a **dup** or **drop**.

Let bare-env : $(\Gamma \vdash ae) \to ctxt$ be a function which takes a derivation and returns the environment at the closest subderivation with a bare expression.

Bare Expressions and Environments

$$\text{bare}\left(x\right), \text{bare}\left(\lambda x.e\right), \text{bare}\left(\langle e_1, e_2\rangle\right), \text{bare}\left([e_1, e_2]\right) = True$$

$$\text{bare}\left(\textbf{dup}\ \Gamma\ \textbf{in}\ e\right), \text{bare}\left(\textbf{drop}\ \Gamma\ \textbf{in}\ e\right) = False$$

$$\text{bare-env}\left(\frac{\mathcal{D}}{\Gamma \vdash ae}\right) = \begin{cases} \Gamma & \text{when bare}\left(ae\right) \\ \text{bare-env}\left(\mathcal{D}\right) & \text{otherwise} \end{cases}$$

Lemma 3.4 (Bare Environments include Sharp Free Variables). *If* $\Gamma \vdash ae$ *and* bare(ae) *then* $\mathit{fv}^{\sharp}\left(\text{erase}\left(ae\right)\right) \sqsubseteq \Gamma$

Proof. By casework on ae.

Case Non-Pair: $ae \neq \langle ae_1, ae_2 \rangle$ then since $\text{bare}\,(ae)$ we know that $\text{erase}\,(ae) \neq \langle ae_1, ae_2 \rangle$ as well. Thus $\text{fv}'\,(\text{erase}\,(ae)) = \text{fv}\,(\text{erase}\,(ae))$ so we can use Lemma 3.3.

Case Pair: $ae = \langle ae_1, ae_2 \rangle$ then we have $\dfrac{\Gamma_1 \vdash ae_1 \qquad \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2 \rangle}$ so by Lemma 3.3 we have $\text{fv}\,(\text{erase}\,(ae_1)) \sqsubseteq \Gamma_1$ and $\text{fv}\,(\text{erase}\,(ae_2)) \sqsubseteq \Gamma_2$. By lemma 3.2 we can combine these inequalities to get $\text{fv}\,(\text{erase}\,(ae_1)) + \text{fv}\,(\text{erase}\,(ae_2)) \sqsubseteq \Gamma_1 + \Gamma_2$. Thus $\text{fv}'\,(\text{erase}\,(ae)) \sqsubseteq \Gamma_1 + \Gamma_2$. □

We can then prove the insertion algorithm is optimal in the sense that it achieves these lower bounds:

Lemma 3.5 (Infer requires Free Variables)**.** *If* $\text{infer}\,(e)$ *returns a derivation for* $\Gamma \vdash ae$ *then* $\Gamma = fv\,(e)$

Proof. Straightforward induction on e, noting that if Γ_1, Γ_2 are multisets which satisfy set-invariants, then so does $\Gamma_1 \sqcup \Gamma_2$. □

Lemma 3.6 (Infer requires Sharp Free Variables)**.** *bare-env* $(\text{infer}\,(e)) = fv'\,(e)$

Proof. Casework on e. Let $\mathcal{D} :: \Gamma \vdash ae = \text{infer}\,(e)$.

Case Non-Pair: If $e \neq \langle e_1, e_2 \rangle$ then $\text{bare}\,(ae)$ and $\text{bare-env}\,(\mathcal{D}) = \Gamma$ and $\text{fv}'\,(e) = \text{fv}\,(e)$ so we can use Lemma 3.5.

Case Pair: If $e = \langle e_1, e_2 \rangle$ then let $\mathcal{D}_i :: \Gamma_i \vdash ae_i = \text{infer}\,(e_i)$.

$$\mathcal{D} = \dfrac{\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2 \rangle}}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{dup}\ \Gamma_1 \sqcap \Gamma_2\ \textbf{in}\ \langle ae_1, ae_2 \rangle}.$$

In this derivation, $\text{bare-env}\,(\mathcal{D}) = \Gamma_1 + \Gamma_2$. By Lemma 3.5, $\Gamma_i = \text{fv}\,(e_i)$. Thus, $\text{bare-env}\,(\mathcal{D}) = \text{fv}'\,(\langle e_1, e_2 \rangle)$ □

Since infer maps recursively over an AST, Lemmas 3.4 and 3.6 tell us that the inference algorithm produces the smallest possible environment at each bare subexpression. We can state this as a theorem:

Theorem 3.2 (Context Minimization)**.** $\text{infer}\,(e)$ *produces a derivation whose contexts at each bare expression are minimal compared to all possible other valid derivations which erase to* e.

Proof. By Contradiction. Assume there exists a valid alternative derivation with smaller bare environment at a subderivation. Lemmas 3.4 and 3.6 show that this is impossible. □

3.3.3 Constraint Optimality

In order for the inference algorithm to interact well with a static type system, it is also key that it never insert a dup or a drop of a variable when avoiding dups or drops entirely could be possible. For example, inserting a **dup x** immediately followed by a **drop x** doesn't affect the live variable environment at any code point, so it is consistent with Theorem 3.2. However, it imposes Dup and Drop type class constraints on the type of x which might have been avoided if x were not duplicated or dropped anywhere else in the code. An inference algorithm which imposed such unnecessary constraints would return frustratingly restrictive types.

Thus, in the arguments that follow we will show that if a variable can avoid being duplicated or dropped in a well-formed annotated expression, then our algorithm avoids doing so.

We begin by establishing some technical conditions on environments that force a well-formed derivation to resort to dropping or duplicating a variable. This will later allow us to argue that other derivations which satisfy these conditions must duplicate or drop the same variables that the inference algorithm does.

Lemma 3.7 (Forced Drop)**.** *If* $\Gamma \vdash ae$ *and* $\Gamma(x) \geq 1$ *and* $x \notin fv(\text{erase}(ae))$*,*
then ae *contains a subterm* $\textbf{drop}\ \Gamma'\ \textbf{in}\ ae_s$ *where* $x \in \Gamma'$*.*

Proof. Induction on $\Gamma \vdash ae$. See appendix A, page 65 for details. □

Lemma 3.8 (Forced Dup)**.** *If* $\Gamma \vdash ae$ *and* $\Gamma(x) \leq 1$ *and there exists a subderivation* $\mathcal{D}_s :: \Gamma_s \vdash ae_s$ *of* $\Gamma \vdash ae$ *with* $\Gamma_s(x) \geq 2$*,*
then ae *contains a subterm* $\textbf{dup}\ \Gamma'\ \textbf{in}\ ae_s$ *where* $x \in \Gamma'$

Proof. Induction on $\Gamma \vdash ae$. See appendix A, page 66 for details. □

Now, by examining the locations at which the infer algorithm inserts dups and drops, we can use the results in the previous section about the minimal environments a derivation must include to show that the conditions in Lemmas 3.7 and 3.8 are satisfied by any other valid derivation, and thus that any other derivation will include analogous duplications or drops.

Lemma 3.9 (No Unnecessary Drops)**.** *Let* $\mathcal{D} :: \Gamma \vdash ae = \text{infer}(e)$*. If* ae *contains a subterm* $\textbf{drop}\ \Gamma_d\ \textbf{in}\ ae_s$ *with* $x \in \Gamma_d$ *then any other well-formed* ae' *with* $\text{erase}(ae') = e$ *contains a subterm* $\textbf{drop}\ \Gamma_d'\ \textbf{in}\ ae_s'$ *with* $x \in \Gamma_d'$*.*

Proof. If we look back at the definition of the inference algorithm, we can see that we only insert drop operations inside subexpressions of certain forms. We can thus consider the shapes of locations where we might insert a Drop. There are two cases:

Case Lam: The Drop appears under a lambda and $\mathcal{D}$ contains a subderivation $\mathcal{D}_1 :: \Gamma_1 \vdash \lambda x.\textbf{drop}\ x\ \textbf{in}\ ae_s$.

By the L-Abs and L-Drop rule, ae_s is well-formed under an environment with no x, so by Lemma 3.3 $x \notin \text{fv}(\text{erase}(ae_s))$. Then, by the L-Abs rule and Lemma 3.7, any annotation of $\text{erase}\,(\lambda x.ae_s)$ must contain a subterm $\textbf{drop}\ \Gamma_d^{'}\ \textbf{in}\ ae_s^{'}$ with $x \in \Gamma_d^{'}$.

Case With: The Drop appears under a "with" form and WLOG $\mathcal{D}$ contains $\mathcal{D}_1 :: \Gamma_1 \vdash ae_1$ where $ae_1 = [\textbf{drop}\ \Gamma_d\ \textbf{in}\ ae_s, ae_{12}]$.

By Lemma 3.5, $\Gamma_1\,(x) \leq 1$, so by the L-With and L-Drop rule ae_s is type-able in a context with no x so by Lemma 3.3, $x \notin \text{fv}\,(\text{erase}\,(ae_s))$. By Theorem 3.2, Γ_1 is the smallest environment that any annotation of $\text{erase}\,(ae_1)$ could be well-formed under. However, from the definition of the infer algorithm, $x \in \Gamma_1$ because $x \in \Gamma_d$. This means that any annotation of $\text{erase}\,(ae_1)$ must have a well-formedness derivation $\Gamma_1^{'} \vdash ae_1^{'}$ with $x \in \Gamma_1^{'}$ so by Lemma 3.7 and the L-With rule this annotation must contain a $\textbf{drop}\ \Gamma_d^{'}\ \textbf{in}\ ae_s^{'}$ subterm with $x \in \Gamma_d^{'}$. □

Lemma 3.10 (No Unnecessary Dups)**.** *Let* $\mathcal{D} :: \Gamma \vdash ae = \text{infer}\,(e)$*. If* ae *contains a subterm* $\textbf{dup}\ \Gamma_d\ \textbf{in}\ ae_s$ *with* $x \in \Gamma_d$ *then any other* ae' *with* $\text{erase}\,(ae') = e$ *and* $\Gamma^{'} \vdash ae'$ *for* $\Gamma^{'}\,(x) \leq 1$ *contains a subterm* $\textbf{dup}\ \Gamma_d^{'}\ \textbf{in}\ ae_s^{'}$ *with* $x \in \Gamma_d^{'}$.

Proof. Consider the locations that the infer algorithm might insert a Dup.

The Dup must appear above a pair and $\mathcal{D}$ contains a subderivation $\mathcal{D}_1 :: \Gamma_1 \vdash ae_1$ where $ae_1 = \textbf{dup}\ \Gamma_d\ \textbf{in}\ \langle ae_{11}, ae_{12}\rangle$.

$$\mathcal{D}_1 = \frac{\Gamma_1 + \Gamma_d \vdash \langle ae_{11}, ae_{12}\rangle}{\Gamma_1 \vdash \textbf{dup}\ \Gamma_d\ \textbf{in}\ \langle ae_{11}, ae_{12}\rangle}\ .$$

We know from Theorem 3.2 that any other annotation of $\text{erase}\,(\langle ae_{11}, ae_{12}\rangle)$ must be well-formed in an environment larger than $\Gamma_1 + \Gamma_d$, which by inversion of $\mathcal{D}_1$ contains at least two copies of x. Since any annotation of e must include an annotation of $\text{erase}\,(\langle ae_{11}, ae_{12}\rangle)$, we know from Lemma 3.8 that any annotation must contain a $\textbf{dup}\ \Gamma_d^{'}\ \textbf{in}\ ae_s^{'}$ subterm with $x \in \Gamma_d^{'}$ □

The above two lemmas can then be glued together.

Definition 3.2. Let $\text{dupvars}\,(ae)$ be the set (not multiset) of all variables x for which a subterm $\textbf{dup}\ \Gamma\ \textbf{in}\ ae_1$ with $x \in \Gamma$ occurs in ae. Let $\text{dropvars}\,(ae)$ be the set of all variables x for which a subterm $\textbf{drop}\ \Gamma\ \textbf{in}\ ae_1$ with $x \in \Gamma$ occurs in ae.

Theorem 3.3 (Minimal Dup/Drop Usage)**.** *Let* $\mathcal{D} :: \cdot \vdash ae = \text{infer}\,(e)$ *for a closed expression* e*. For any other well formed closed annotation* ae' *with* $\text{erase}\,(ae') = e$*,* $\mathit{dupvars}\,(ae) \subseteq \mathit{dupvars}\,(ae')$ *and* $\mathit{dropvars}\,(ae) \subseteq \mathit{dropvars}\,(ae')$ *.*

Chapter 4

The Clamp Type System

The calculus λ_{cl} captures the type system of the Clamp Programming Language and is based off of System-F [Girard, 1972]. To System-F, two fundamental extensions are made. First, environment handling and variable binding are treated linearly as in standard linear lambda calculi or linear logics [Abramsky, 1993, Girard, 1987]. Second, to allow for substructural dup and drop operations, type class constraints are added to type schemes as in other calculi that support qualified or predicated type schemes [Stuckey and Sulzmann, 2005, Smith, 1994, Wadler and Blott, 1989]. The λ_{cl} type system also shares many similarities with the ${}^{a}\lambda_{ms}$ calculus in Tov and Pucella [2011].

Unlike the more user-friendly Clamp language as implemented and described in Chapter 2, λ_{cl} is presented with first class polymorphism and does not support type inference. By translating the language into a Hindley-Milner framework such as HM(X) however, it is straightforward to add type inference at the cost of first class polymorphism.

4.1 λ_{cl} Syntax and Typing

4.1.1 Syntax

In figure 4.1 on the following page we present the syntax for λ_{cl}. Dup and drop operations are explicit in this language, and it can serve as a target for the insertion algorithm described in chapter 3.

There are standard forms for variables, lambda-abstraction and application. Note however that lambdas are annotated with a substructural qualifier, since λ_{cl} has four distinct arrow types. Type abstractions also specify the type class constraints they abstract over, and include a value restriction to avoid forming closures inside type abstractions (the value forms are given in figure 4.11 on page 45). This means that, unlike lambdas, type abstractions do not need to be annotated with substructural qualifiers, and also interact nicely with references.

There is no need to include let-bindings since they can be encoded using linear-lambdas. The **letp** $(x_1, x_2) = e$ form binds both components of a linear pair with one use, since the standard projection operators could only retrieve one component at a time. Using projection operators in

$$\textbf{let } x = \textbf{fst } z \textbf{ in let } y = \textbf{snd } z \textbf{ in } x + y$$

λ_{cl} Expressions

$$\begin{aligned}
e ::=\ & x \mid \lambda^{aq}\,(x : \tau)\,.e \mid e_1\ e_2 \mid \Lambda\overline{\alpha_i}\,[P]\,.v \mid e\,[\overline{\tau_i}] \\
& \mid (e_1, e_2) \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid () \\
& \mid \mathbf{letp}\ \ (x_1, x_2) = e\ \mathbf{in}\ e_2 \\
& \mid \mathbf{match}\ e\ \mathbf{with\ inl}\ x_1 \to e_1 ; \mathbf{inr}\ x_2 \to e_2 \\
& \mid \ell \mid \mathbf{new}^{\mathbf{rq}}\ e \mid \mathbf{release}^{\mathbf{rq}}\ e \mid \mathbf{swap}^{\mathbf{rq}}\ e_1\ \mathbf{with}\ e_2 \\
& \mid \mathbf{dup}\ e_1\ \mathbf{as}\ x_1, x_2\ \mathbf{in}\ e_2 \mid \mathbf{drop}\ e_1\ \mathbf{in}\ e_2 \\
\text{rq} ::=\ & \mathbf{s}\ (\text{strong}) \mid \mathbf{w}\ (\text{weak}) \\
\text{aq} ::=\ & \mathbf{U}\ (\text{unlimited}) \mid \mathbf{R}\ (\text{relevant}) \mid \mathbf{A}\ (\text{affine}) \mid \mathbf{L}\ (\text{linear})
\end{aligned}$$

Figure 4.1: λ_{cl} Syntax

would violate linearity if z were linear. **inl** e and **inr** e are introduction injections for sums (additive disjunction), and the **match** form eliminates the sum by case analysis as usual. A qualified "with" $\&^{aq}$ form would be straightforward to add to the language, but can also be encoded using sums and lambdas.

The $\mathbf{new}^{\mathbf{rq}}$ e and $\mathbf{release}^{\mathbf{rq}}$ e forms are the introduction and elimination forms for references. They come in two different forms depending on the parameter rq which describes whether the reference supports strong update. $\mathbf{swap}^{\mathbf{rq}}$ e_1 **with** e_2 allows linear access to a reference by exchanging its contents for new contents. The ℓ are store locations which are present at runtime but not written by the programmer.

dup e_1 **as** x_1, x_2 **in** e_2 allows for the substructural operation of aliasing an expression e_1 and binding the two copies. **drop** e_1 **in** e_2 similarly disposes the expression e_1. Though these substructural operations are usually thought of as operating on variables, by extending them to work over expressions it will be easier to define a small-step substitution semantics later.

The types in λ_{cl} are given in figure 4.2 on the facing page. Clamp contains the standard sum and product types corresponding to the $\oplus$ and $\otimes$ connectives in linear logic. It also contains both strong and weak reference types. As mentioned earlier, since arrows are annotated by their substructural qualifier aq, λ_{cl} has 4 distinct arrow types also annotated by aq qualifiers. To incorporate type classes, type schemes encapsulate constraints on their type variables. P is used to denote a set of atomic Predicate constraints Pred, which are predicate constructors K applied to a type. For the sake of our current analysis, K is either a Dup or a Drop.

Since the type system for λ_{cl} is linear, it makes use of careful manipulation of variable and location typing contexts as defined in figure 4.3 on the next page. Variable type contexts Γ are sets (not multisets) of variable-to-type bindings. Location type contexts Σ are considered

λ_{cl} **Types**

$$\begin{aligned}
\tau &::= \alpha \mid \tau_1 \xrightarrow{aq} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathrm{ref}^{\mathrm{rq}}\tau \mid \forall\overline{\alpha_i}\,[P]\,.\tau \\
P &::= \mathrm{Pred}_1, \ldots, \mathrm{Pred}_n \\
\mathrm{Pred} &::= K\tau \\
K &::= \mathrm{Dup} \ \mid \mathrm{Drop}
\end{aligned}$$

Figure 4.2: λ_{cl} Types

Contexts

$$\begin{aligned}
\Gamma &::= x_1 : \tau_1, \ldots, x_n : \tau_n \\
\Sigma^s &::= \ell_1 \mapsto_{\mathbf{s}} \tau_1, \ldots, \ell_n \mapsto_{\mathbf{s}} \tau_n \\
\Sigma^w &::= \ell_1 \mapsto_{\mathbf{w}}^{j_1} \tau_1, \ldots, \ell_n \mapsto_{\mathbf{w}}^{j_n} \tau_n \qquad j_i > 0 \\
\Sigma &::= \Sigma^s, \Sigma^w \qquad \mathrm{Dom}\,(\Sigma^s) \cap \mathrm{Dom}\,(\Sigma^w) = \emptyset
\end{aligned}$$

Figure 4.3: Variable and Location Typing Contexts

as sets containing both strong location bindings Σ^s and reference-counted weak location bindings Σ^w. Weak location bindings are annotated with a reference count to track their usages so that we know when we can deallocate them.

All variables and type variables are assumed to have distinct names in λ_{cl} to avoid issues with shadowing.

Since they are sets of bindings, both kinds of contexts can also be viewed as partial maps from variables or locations to types. Exchange properties are thus implicit for both contexts. It is also assumed that all environment are well formed in that a variable is bound at most once, and we will use the Dom function to refer to the set of variables or labels whose bindings are present in a context.

$$\mathrm{Dom}\left(\overline{\ell_i \mapsto_{\mathbf{rq}}^{j_i} \tau_i}\right) = \left\{\overline{\ell_i}\right\}$$

$$\mathrm{Dom}\,(\overline{x_i : \tau_i}) = \{\overline{x_i}\}$$

Because Γ and Σ are linear, we can define operations to join them in figure 4.4 on the following page. However, the join operations $\circ$ and $+$ are only well-defined when the two component environments are compatible, which is denoted by $\Gamma_1 \smile \Gamma_2$ and $\Sigma_1 \smile \Sigma_2$. Two

Context Operations

$$
\begin{aligned}
\Gamma_1 \smile \Gamma_2 &\equiv \mathrm{Dom}\,(\Gamma_1) \cap \mathrm{Dom}\,(\Gamma_1) = \emptyset \\
\Sigma_1 \smile \Sigma_2 &\equiv \mathrm{Dom}\,(\Sigma_1^s) \cap \mathrm{Dom}\,(\Sigma_2) = \emptyset \text{ and } \mathrm{Dom}\,(\Sigma_2^s) \cap \mathrm{Dom}\,(\Sigma_1) = \emptyset \text{ and} \\
&\qquad \forall \ell \in \mathrm{Dom}\,(\Sigma_1^w) \cap \mathrm{Dom}\,(\Sigma_2^w) \,.\, \Sigma_1\,(\ell) = \Sigma_2\,(\ell) \\
\Gamma_1 \circ \Gamma_1 &\equiv \Gamma_1, \Gamma_2 \text{ when } \Gamma_1 \smile \Gamma_2 \\
\Sigma_1 + \Sigma_2 &\equiv \Sigma_1^s, \Sigma_2^s, \\
&\qquad \left\{\ell \mapsto_{\mathbf{w}}^{j} \tau \mid \ell \mapsto_{\mathbf{w}}^{j} \tau \in \Sigma_1^w \wedge \ell \notin \mathrm{Dom}\Sigma_2^w \text{ or } \ell \mapsto_{\mathbf{w}}^{j} \tau \in \Sigma_2^w \wedge \ell \notin \mathrm{Dom}\Sigma_1^w\right\}, \\
&\qquad \left\{\ell \mapsto_{\mathbf{w}}^{j_1+j_2} \tau \mid \ell \mapsto_{\mathbf{w}}^{j_1} \tau \in \Sigma_1, \ell \mapsto_{\mathbf{w}}^{j_2} \tau \in \Sigma_2\right\}, \\
&\qquad \text{when } \Sigma_1 \smile \Sigma_2
\end{aligned}
$$

Figure 4.4: Context Compatibility and Join

variable contexts are compatible so long as they are disjoint, so that one never ends up with inconsistent or multiple bindings. Two store contexts are compatible if the strong locations are disjoint and the weak locations at least agree on their types. Joining variable contexts appends the two sets of bindings together, while joining location contexts also involves combining the reference counts of any shared weak locations. In many ways, the operations for combining contexts resemble those in separation algebras.

We also extend our constraint judgments onto contexts in the usual way by lifting the constraints onto the types referred to in the contexts, and define shorthand abbreviations for constrained contexts. These constraints on contexts are defined in figure 4.5 on the next page.

4.1.2 Typing Rules

The expression typing judgment assigning an expression e a type τ takes the form of a 5 way relation including variable and location contexts Γ and Σ, as well as a constraint context P of type class predicates. The location contexts Σ are not used in typing input programs, but are used in tracking runtime locations ℓ.

$$\boxed{\text{Expression Typing: } P; \Gamma; \Sigma \vdash e : \tau}$$

The inference rules for this typing relation are given in figures 4.6 and 4.7 on page 40. Well formedness conditions for type variables appearing in types and on contexts are left implied. Similarly consistency conditions $\Sigma_1 \smile \Sigma_2$ and $\Gamma_1 \smile \Gamma_2$ are implied whenever contexts are combined.

The core language typing rules are a natural extension of System-F to support type class

Context Constraints

$$K\,\Gamma \iff K\tau \text{ for all } x:\tau\in\Gamma$$

$$K\,\Sigma \iff K\,(\text{ref}^{\text{s}}\tau) \text{ for all } \ell\mapsto_{\mathbf{s}}\tau\in\Sigma \text{ and } K\,(\text{ref}^{\text{w}}\tau) \text{ for all } \ell\mapsto_{\mathbf{w}}^{i}\tau\in\Sigma$$

$$\text{Constrain}^{U}(\Gamma,\Sigma)\equiv \text{Dup }\Gamma,\Sigma,\ \text{Drop }\Gamma,\Sigma$$

$$\text{Constrain}^{R}(\Gamma,\Sigma)\equiv \text{Dup }\Gamma,\Sigma$$

$$\text{Constrain}^{A}(\Gamma,\Sigma)\equiv \text{Drop }\Gamma,\Sigma$$

$$\text{Constrain}^{L}(\Gamma,\Sigma)\equiv \mathit{nil}$$

Figure 4.5: Context Constraints

λ_{cl}-Typing Core

Cl-Var

$$\frac{}{P;x:\tau;\cdot\vdash x:\tau}$$

Cl-TAbs

$$\frac{P_1,P_2;\Gamma;\Sigma\vdash v:\tau \qquad \text{Dom}\,(P_2)\subset\overline{\alpha_i}}{P_1;\Gamma;\Sigma\vdash \Lambda\overline{\alpha_i}\,[P_2]\,.v:\forall\overline{\alpha_i}\,[P_2]\,.\tau}$$

Cl-Tapp

$$\frac{P_1;\Gamma;\Sigma\vdash e:\forall\overline{\alpha_i}\,[P_2]\,.\tau \qquad P_1\Vdash P_2\overline{\{\tau_i/\alpha_i\}}}{P_1;\Gamma;\Sigma\vdash e\,[\overline{\tau_i}]:\tau\overline{\{\tau_i/\alpha_i\}}}$$

Cl-Lam

$$\frac{P;\Gamma,x:\tau_1;\Sigma\vdash e:\tau_2 \qquad P\Vdash \text{Constrain}^{aq}(\Gamma,\Sigma)}{P;\Gamma;\Sigma\vdash\lambda^{aq}\,(x:\tau_1)\,.e:\tau_1\xrightarrow{aq}\tau_2}$$

Cl-App

$$\frac{P;\Gamma_1;\Sigma_1\vdash e_1:\tau_2\xrightarrow{aq}\tau \qquad P;\Gamma_2;\Sigma_2\vdash e_2:\tau_2}{P;\Gamma_1\circ\Gamma_2;\Sigma_1+\Sigma_2\vdash e_1\ e_2:\tau}$$

Cl-Dup

$$\frac{P;\Gamma_1;\Sigma_1\vdash e_1:\tau_1 \qquad P;\Gamma_2,x_1:\tau_1,x_2:\tau_1;\Sigma_2\vdash e_2:\tau_2 \qquad P\Vdash \text{Dup }\tau_1}{P;\Gamma_1\circ\Gamma_2;\Sigma_1+\Sigma_2\vdash \textbf{dup }e_1\textbf{ as }x_1,x_2\textbf{ in }e_2:\tau_2}$$

Cl-Drop

$$\frac{P;\Gamma_1;\Sigma_1\vdash e_1:\tau_1 \qquad P;\Gamma_2;\Sigma_2\vdash e_2:\tau_2 \qquad P\Vdash \text{Drop }\tau_1}{P;\Gamma_1\circ\Gamma_2;\Sigma_1+\Sigma_2\vdash \textbf{drop }e_1\textbf{ in }e_2:\tau_2}$$

Figure 4.6: λ_{cl} Expression Typing

λ_{cl}-Typing Data

Cl-Pair
$$\frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \tau_1 \qquad P;\Gamma_2;\Sigma_2 \vdash e_2 : \tau_2}{P;\Gamma_1 \circ \Gamma_2;\Sigma_1 + \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

Cl-Inl
$$\frac{P;\Gamma;\Sigma \vdash e : \tau_1}{P;\Gamma;\Sigma \vdash \textbf{inl } e : \tau_1 + \tau_2}$$

Cl-Inr
$$\frac{P;\Gamma;\Sigma \vdash e : \tau_1}{P;\Gamma;\Sigma \vdash \textbf{inr } e : \tau_2 + \tau_1}$$

Cl-Unit
$$\frac{}{P;\cdot;\cdot \vdash () : \text{unit}}$$

Cl-Letp
$$\frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \tau_{11} \times \tau_{12} \qquad P;\Gamma_2, x_1 : \tau_{11}, x_2 : \tau_{12};\Sigma_2 \vdash e_2 : \tau_2}{P;\Gamma_1 \circ \Gamma_2;\Sigma_1 + \Sigma_2 \vdash \textbf{letp } (x_1, x_2) = e_1 \textbf{ in } e_2 : \tau_2}$$

Cl-Match
$$\frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \tau_{11} + \tau_{12} \qquad P;\Gamma_2, x_{21} : \tau_{11};\Sigma_2 \vdash e_{21} : \tau_2 \qquad P;\Gamma_2, x_{22} : \tau_{12};\Sigma_2 \vdash e_{22} : \tau_2}{P;\Gamma_1 \circ \Gamma_2;\Sigma_1 + \Sigma_2 \vdash \textbf{match } e_1 \textbf{ with inl } x_{21} \rightarrow e_{21}; \textbf{inr } x_{22} \rightarrow e_{22} : \tau_2}$$

λ_{cl}-Typing Ref

Cl-LocW
$$\frac{}{P;\cdot;\ell \mapsto^1_{\mathbf{w}} \tau \vdash \ell : \text{ref}^{\text{w}}\tau}$$

Cl-LocS
$$\frac{}{P;\cdot;\ell \mapsto_{\mathbf{s}} \tau \vdash \ell : \text{ref}^{\text{s}}\tau}$$

Cl-New
$$\frac{P;\Gamma;\Sigma \vdash e : \tau}{P;\Gamma;\Sigma \vdash \textbf{new}^{\textbf{rq}}\ e : \text{ref}^{\text{rq}}\tau}$$

Cl-ReleaseW
$$\frac{P;\Gamma;\Sigma \vdash e : \text{ref}^{\text{rq}}\tau}{P;\Gamma;\Sigma \vdash \textbf{release}^{\textbf{w}}\ e : \text{unit} + \tau}$$

Cl-ReleaseS
$$\frac{P;\Gamma;\Sigma \vdash e : \text{ref}^{\text{s}}\tau}{P;\Gamma;\Sigma \vdash \textbf{release}^{\textbf{s}}\ e : \tau}$$

Cl-SwapW
$$\frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \text{ref}^{\text{rq}}\tau \qquad P;\Gamma_2;\Sigma_2 \vdash e_2 : \tau}{P;\Gamma_1 \circ \Gamma_2;\Sigma_1 + \Sigma_2 \vdash \textbf{swap}^{\textbf{w}}\ e_1 \textbf{ with } e_2 : \text{ref}^{\text{rq}}\tau \times \tau}$$

Cl-SwapS
$$\frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \text{ref}^{\text{s}}\tau_1 \qquad P;\Gamma_2;\Sigma_2 \vdash e_2 : \tau_2}{P;\Gamma_1 \circ \Gamma_2;\Sigma_1 + \Sigma_2 \vdash \textbf{swap}^{\textbf{s}}\ e_1 \textbf{ with } e_2 : \text{ref}^{\text{s}}\tau_2 \times \tau_1}$$

Figure 4.7: λ_{cl} Expression Typing (continued)

constraints. A syntactic restriction (similar to the context reduction restrictions in Haskell 98) is imposed on the form of constraints we can abstract over in the Cl-Tabs rule, meaning that type abstractions can only constrain the type variables they abstract over, and not compound or unrelated types. This makes the meta-theory much easier, and the restriction is specified using the Dom operator applied to a set of predicates P, which returns the set of types which are actually constrained by a set of predicates.

$$\text{Dom}\left(\overline{K_i \tau_i}\right) = \overline{\tau_i}$$

In addition, the different substructural arrows impose the corresponding substructural constraints on both the variable and store contexts they close over in the Cl-Lam rule using the entailment relation $\Vdash$. The entailment relation $P_1 \Vdash P_2$ can be read "The constraints in P_2 are satisfied given the assumptions in P_1" and the relation is defined in section 4.2. Type abstractions do not need to impose constraints on their contexts as lambdas do because of the value restriction.

The dup and drop operations constrain the types of their arguments in the natural way. The only two places where substructural constraints are externally imposed by the type system are thus in the dup/drop and arrow typing rules.

Since λ_{cl} supports both strong and weak references with different substructural properties, there are a variety of typing rules governing their usage. The runtime store locations ℓ are not tagged with a qualifier rq, so whether a location is treated as strong or weak depends on the typing context Σ. The **swap** operation returns both an updated reference and the old contents, so we have it return a pair which can be destructed later. Since both weak and strong forms of reference cell operations are provided, it is sound to apply the weak ones to both strong and weak references. The $\textbf{release}^{\textbf{rq}}$ e forms are used to deallocate a reference cell and possibly retrieve its contents. Notably, in the case of a weak reference, since the contents could be linear one way of ensuring linearity while allowing for aliasing is to return the contents of the reference when the last alias to the cell is released, and unit otherwise.

4.2 Typeclass Instances

The key relation on predicates in our type system is entailment, which describes when one set of predicates can be inferred from another in the context of the background instance environment Γ^{is}.

$$\boxed{\text{Entailment: } P_1 \Vdash P_2}$$

For instance, entailment allows our type system to derive that int $\times$ unit is duplicable because int and unit are. The rules for entailment presented below are inspired by Jones [1995].

As a definition, one set of predicates entails another when one can derive each predicate in the latter from the former.

$$\boxed{P_1 \Vdash \overline{\text{Pred}_i} \iff \overline{P_1 \Vdash \text{Pred}_i}}$$

Then, one can give the rules for deriving entailments for single predicates in figure 4.8 on the next page.

Predicates

$$\textsc{Pred-Elt}\quad \frac{\text{Pred} \in P}{P \Vdash \text{Pred}}$$

$$\textsc{Pred-Inst}\quad \frac{P_1 \Vdash P_2 \qquad (\text{instance } P_2 \implies \text{Pred}) \in \Gamma^{\text{is}}\overline{\{\tau_i/\alpha_i\}}}{P_1 \Vdash \text{Pred}} \qquad \textsc{Pred-Sch}\quad \frac{P_1, P_2 \Vdash K\,\tau}{P_1 \Vdash K\ (\forall\overline{\alpha_i}\,[P_2]\,.\tau)}$$

Figure 4.8: Predicate Entailment

Any predicate already in the assumed context is entailed by the Pred-Elt rule. The Pred-Inst rule allows one to make use of instances in Γ^{is} . Type abstractions are the only types that cannot be covered by the Pred-Inst rule since they are not built from basic type constructors, so the Pred-Sch rule defines how their constraints can be derived.

The set of derivable type class instances depends directly on the set of base instances in Γ^{is}, so must define the base instances carefully to preserve substructural properties. In particular, the instances in Γ^{is} are limited to a specific syntactic structure. This makes the theory simpler, and is easy to check in λ_{cl} where the set of instances Γ^{is} is given and fixed.

$$is ::= \text{instance } \overline{K_i\alpha_i} \implies K\,(\text{TCon } \overline{\alpha_j}) \qquad \overline{\alpha_i} \subset \overline{\alpha_j}$$
$$\Gamma^{\text{is}} ::= \{is_1, is_2, \cdots, is_n\}$$

Then, to define the substructural properties of the built-in types in λ_{cl}, figure 4.9 on the facing page gives the base instance rules in Γ^{is}.

On the core types, we need restrictions on duplicating and dropping containers. Since pairs and sums contain values which might be copied or ignored along with the container, their instance rules depend on the types of their components. Arrows impose constraints on their closure environments when they are assigned a qualifier during expression typing, so the instance rules for arrows depend only on the arrow qualifier. In particular, the substructural properties of arrows do not depend on their domain and range, since for instance if y is bound to a string, then duplicating a function $\lambda^{\mathbf{R}}x.\text{newfile}\,(y)$ of type $\forall\alpha\,[]\,.\alpha \xrightarrow{\mathbf{R}} \text{file}$ does not require duplicating files but does require duplicating y of type string.

Given this closed set of instances in Γ^{is}, there is at most one way to derive a Dup or Drop constraint for a type, since each instance has a conclusion with different head type constructor, and the Pred-Sch rule is the only way to infer constraints on type abstractions.

Dealing correctly with references is more subtle, as seen in λ^{URAL} [Ahmed et al., 2005]. However, we will see that using Dup and Drop type classes alongside reference counting semantics allows the rules in λ^{URAL} to be expressed very simply.

In λ^{URAL}, for kinds of references are provided, one for each point on the URAL lattice. For our purposes, the complete set of restrictions for substructural references in λ^{URAL} can

Base Instances

$$
\begin{array}{rcl@{\qquad}rcl}
\text{Dup } a, \text{Dup } b & \Longrightarrow & \text{Dup } (a \times b) & \text{Drop } a, \text{Drop } b & \Longrightarrow & \text{Drop } (a \times b) \\
\text{Dup } a, \text{Dup } b & \Longrightarrow & \text{Dup } (a + b) & \text{Drop } a, \text{Drop } b & \Longrightarrow & \text{Drop } (a + b) \\
nil & \Longrightarrow & \text{Dup } \left(a \xrightarrow{U} b\right) & nil & \Longrightarrow & \text{Drop } \left(a \xrightarrow{U} b\right) \\
nil & \Longrightarrow & \text{Dup } \left(a \xrightarrow{R} b\right) & nil & \Longrightarrow & \text{Drop } \left(a \xrightarrow{A} b\right) \\
nil & \Longrightarrow & \text{Dup unit} & nil & \Longrightarrow & \text{Drop unit} \\
nil & \Longrightarrow & \text{Dup } (\text{ref}^{\text{w}} a) & \text{Drop } a & \Longrightarrow & \text{Drop } (\text{ref}^{\text{rq}} a)
\end{array}
$$

Figure 4.9: Base Substructural Type Class Instances in Γ^{is}

be broken down as thus:

- A and L references support strong update since they cannot be aliased, while U and R only support weak update
- U and A references cannot store R and L data since these references can be dropped, leaving the undroppable data hanging. R and L refs can store R and L data
- U and R references do not support the "free : ref a -> a" elimination form, which retries the data while consuming a reference, since the actual storage cell may be aliased. A and L refs do support the "free" elimination form

Interpreting substructural types in terms of type classes simplifies these rules. Rather than having 4 types of refs, of which strong updates are tied tightly to A and L refs, in Clamp we have only strong references and weak references. Interpreting the above rules in our framework yields the following restrictions:

- Strong references cannot be duplicated
- References which support drop can only store droppable data
- Strong references support "free" (The **release**$^{\mathbf{s}}$ e form in λ_{cl}) as in URAL.
- Weak references, which may be aliased, support a version of free with type: $\text{ref}^{\text{rq}} a \xrightarrow{\mathbf{U}} a + \text{unit}$. This free only returns the contents of type a when the last copy of the reference is freed, and otherwise returns unit.

The above four rules capture the restrictions given for λ^{URAL} reference and further increase the expressiveness of the system by splitting out weak and strong references and allowing for the deallocation of weak references. They are expressed in λ_{cl} with 2 short type class instances and 2 typing judgments. In this system, rather than having the user specify substructural properties, he can merely specify the more natural notion of whether he would like a reference cell to support strong updates. Then the type system can automatically infer the maximum possible set of substructural operations allowable on the reference based on its contents. In a full ML or Haskell-like language one can of course declare custom special reference types without one or more of these capabilities.

In λ^{URAL}, references are also classified in terms of whether they support the **read** and **write** operations (in addition to the primitive swap operation). This is very straightforward to translate into the language of type classes. Though this is not included in λ_{cl} or Clamp, to support Read/Write operations one might define type classes with the following instances:

$$\text{Dup } a \implies \text{Readable ref}^{\text{rq}} a \qquad\qquad \text{Drop } a \implies \text{Writable ref}^{\text{rq}} a$$

For simplicity then, the read and write operations can be defined at the user level, and will not be included in the language definition.

4.3 Small Step Semantics

A more realistic semantics for λ_{cl} is given in chapter 5, but a simple small step semantics for λ_{cl} is also necessary to give a type soundness proof. We can thus develop a semantics for λ_{cl} that does not take into account the substructural properties of values directly at runtime, but does so indirectly through the handling of references.

4.3.1 Linear Substitution

The substitution operator can be defined as it is in standard lambda calculus, but its behavior is more specific on well typed terms in a linear language. Ignoring the other components of typing relation for the moment, if $\Gamma \vdash e$ then Γ tells us exactly the free variables in e, so we can use our typing rules to further constrain the substitution operation. For instance, the variable we are substituting out cannot appear in both $\Gamma_1 \vdash e_1$ and $\Gamma_2 \vdash e_2$ if we have $\Gamma_1 \circ \Gamma_2 \vdash e$ since there is an implicit disjointness condition when combining contexts.

Thus linear substitution can be defined to take into account the linear usage of variables in figure 4.10 on the next page, and this will prove useful in the proofs and for intuition for the small step semantics. Throughout this chapter when performing substitution on well typed terms we will be referring to linear substitution.

4.3.2 Small-Step Semantics

The small-step semantics uses execution contexts and a global reference-counted store μ to define a call-by-value single step relation. The runtime structures needed to define this small step relation are given in figure 4.11 on the facing page. Since the store is more of a map than a list, exchange properties are implicitly assumed.

Some auxiliary functions for dealing with reference counts on the store are also useful to define. The function floc given in figure 4.12 on page 46 tells us the multiset of locations

Linear Substitution

$$x\{v/x\} = v \qquad \frac{x \notin \mathrm{fv}(e)}{e\{v/x\} = e}$$

$$\frac{e\{v/x\} = e' \qquad x \neq y}{(\lambda y.e)\{v/x\} = \lambda y.e'}$$

$$\frac{e_1\{v/x\} = e_1' \qquad x \notin \mathrm{fv}(e_2)}{(e_1\ e_2)\{v/x\} = e_1'\ e_2} \qquad \frac{e_2\{v/x\} = e_2' \qquad x \notin \mathrm{fv}(e_1)}{(e_1\ e_2)\{v/x\} = e_1\ e_2'}$$

$$\frac{e\{v/x\} = e' \qquad x \notin \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)}{(\textbf{match}\ e\ \textbf{with inl}\ x_1 \to e_1; \textbf{inr}\ x_2 \to e_2)\{v/x\} = \textbf{match}\ e'\ \textbf{with inl}\ x_1 \to e_1; \textbf{inr}\ x_2 \to e_2}$$

$$\frac{e_1\{v/x\} = e_1' \qquad e_2\{v/x\} = e_2' \qquad x \notin \mathrm{fv}(e)}{(\textbf{match}\ e\ \textbf{with inl}\ x_1 \to e_1; \textbf{inr}\ x_2 \to e_2)\{v/x\} = \textbf{match}\ e\ \textbf{with inl}\ x_1 \to e_1'; \textbf{inr}\ x_2 \to e_2'}$$

$$\cdots$$

Figure 4.10: Linear Substitution

Semantic Structs

$$\begin{aligned} E ::= {} & [\cdot] \mid E\ e \mid v\ E \mid E\,[\overline{\tau_i}] \mid (E, e) \mid (v, E) \mid \textbf{inl}\ E \mid \textbf{inr}\ E \\ & \mid \textbf{match}\ E\ \textbf{with inl}\ x_1 \to e_1; \textbf{inr}\ x_2 \to e_2 \mid \textbf{letp}\ (x_1, x_2) = E\ \textbf{in}\ e \\ & \mid \textbf{new}^{\textbf{rq}}\ E \mid \textbf{release}^{\textbf{rq}}\ E \mid \textbf{swap}^{\textbf{rq}}\ E\ \textbf{with}\ e \mid \textbf{swap}^{\textbf{rq}}\ v\ \textbf{with}\ E \\ & \mid \textbf{dup}\ E\ \textbf{as}\ x_1, x_2\ \textbf{in}\ e \mid \textbf{drop}\ E\ \textbf{in}\ e \\ v ::= {} & \lambda^{aq}(x : \tau).e \mid \Lambda\overline{\alpha_i}\,[P].v \mid (v_1, v_2) \mid \textbf{inl}\ v \mid \textbf{inr}\ v \mid \ell \mid () \\ \mu ::= {} & \ell \mapsto^i v, \mu \mid \cdot \end{aligned}$$

Figure 4.11: Runtime Structures

Free Locations

$$\text{floc}\,(x) = \{\} \qquad \text{floc}\,(\ell) = \{\ell\}$$

$$\text{floc}\,(\lambda^{aq}\,(x:\tau)\,.e) = \text{floc}\,(e) \qquad \text{floc}\,(\Lambda\overline{\alpha_i}\,[P]\,.v) = \text{floc}\,(v)$$

$$\text{floc}\,(e_1\ e_2) = \text{floc}\,(e_1) + \text{floc}\,(e_2) \qquad \text{floc}\,((e_1, e_2)) = \text{floc}\,(e_1) + \text{floc}\,(e_2)$$

$$\text{floc}\,(\textbf{letp}\ \ (x_1, x_2) = e\ \textbf{in}\ e_2) = \text{floc}\,(e) + \text{floc}\,(e_2)$$

$$\text{floc}\,(\textbf{match}\ e\ \textbf{with}\ \textbf{inl}\ x_1 \to e_1; \textbf{inr}\ x_2 \to e_2) = \text{floc}\,(e) + (\text{floc}\,(e_1) \sqcup \text{floc}\,(e_2))$$

$$\cdots$$

Figure 4.12: Free Location Counting

Reference Manipulation

$$\text{incr}\ \ell\ \text{in}\ \ell \mapsto^{j} v, \mu = \ell \mapsto^{j+1} v, \mu \qquad \text{incr}\ \overline{\ell_i}\ \text{in}\ \mu = \overline{\text{incr}\ \ell_i\ \text{in}\ \cdots}\mu$$

$$\text{decr}\ \ell\ \text{in}\ \ell \mapsto^{j} v, \mu = \ell \mapsto^{j-1} v, \mu \quad \text{when}\ j > 1 \qquad \text{decr}\ \overline{\ell_i}\ \text{in}\ \mu = \overline{\text{decr}\ \ell_i\ \text{in}\ \cdots}\mu$$

$$\text{decr}\ \ell\ \text{in}\ \ell \mapsto^{1} v, \mu = \text{decr}\ \ (\text{floc}\,(v))\ \ \text{in}\ \mu$$

Figure 4.13: Reference Count Management

that an expression uses, with multiset operations corresponding to those defined in figure 3.3 on page 25. Note that an expression could use a location more than once, and in this case floc would return multiple copies of that location.

The functions incr ℓ in μ and decr ℓ in μ given in figure 4.13 allow us to increment and decrement reference counts in the heap. Incrementing a location is straightforward, but a decrement must be defined recursively since deallocating the last pointer to a weak reference cell involves decrementing the reference counts of all cells the deallocated contents originally pointed to.

Finally, we can then define the rules for the single step relation in figure 4.14 on the facing page.

$$\boxed{\text{Single Step Relation: } (\mu_1\ ;\ e_1)\ \longmapsto\ (\mu_2\ ;\ e_2)}$$

Linear substitution is used to bind variables to values at runtime, while the **dup** and **drop** operations must also manipulate the reference counts of any weak references inside

Small-Step Semantics Core

$$(\mu\ ;\ (\lambda\,(x:\tau)\,.e)\ \ v) \longmapsto (\mu\ ;\ e\,\{v/x\}) \qquad \text{NS-BetaV}$$

$$(\mu\ ;\ (\Lambda\overline{\alpha_i}\,[P]\,.v)\,[\overline{\tau_i}]) \longmapsto \left(\mu\ ;\ v\overline{\{\tau_i/\alpha_i\}}\right) \qquad \text{NS-BetaT}$$

$$(\mu\ ;\ \mathbf{match}\ \mathbf{inl}\ v\ \mathbf{with}\ \mathbf{inl}\ x_1 \to e_1; \mathbf{inr}\ x_2 \to e_2) \longmapsto (\mu\ ;\ e_1\,\{v/x_1\}) \qquad \text{NS-MatchL}$$

$$(\mu\ ;\ \mathbf{match}\ \mathbf{inr}\ v\ \mathbf{with}\ \mathbf{inl}\ x_1 \to e_1; \mathbf{inr}\ x_2 \to e_2) \longmapsto (\mu\ ;\ e_2\,\{v/x_2\}) \qquad \text{NS-MatchR}$$

$$(\mu\ ;\ \mathbf{letp}\ \ (x_1,x_2) = (v_1,v_2)\ \ \mathbf{in}\ e) \longmapsto (\mu\ ;\ e\,\{v_1/x_1\}\,\{v_2/x_2\}) \qquad \text{NS-Letp}$$

$$(\mu_1\ ;\ E\,[e_1]) \longmapsto (\mu_2\ ;\ E\,[e_2]) \qquad \text{NS-Context}$$

$$\text{when}\ \ (\mu_1\ ;\ e_1) \longmapsto (\mu_2\ ;\ e_2)$$

Small-Step Semantics Substruct

$$(\mu\ ;\ \mathbf{dup}\ v\ \mathbf{as}\ x_1,x_2\ \mathbf{in}\ e) \longmapsto (\text{incr floc}\,(v)\ \ \text{in}\ \mu\ ;\ e\,\{v/x_1\}\,\{v/x_2\}) \qquad \text{NS-Dup}$$

$$(\mu\ ;\ \mathbf{drop}\ v\ \mathbf{in}\ e) \longmapsto (\text{decr floc}\,(v)\ \ \text{in}\ \mu\ ;\ e) \qquad \text{NS-Drop}$$

Small-Step Semantics Ref

$$(\mu\ ;\ \mathbf{new}^{\mathbf{rq}}\ v) \longmapsto \left(\mu, \ell \mapsto^1 v\ ;\ \ell\right) \quad \ell\ \text{fresh} \qquad \text{NS-New}$$

$$\left(\mu, \ell \mapsto^i v_1\ ;\ \mathbf{swap}^{\mathbf{rq}}\ \ell\ \mathbf{with}\ v_2\right) \longmapsto \left(\mu, \ell \mapsto^i v_2\ ;\ (\ell, v_1)\right) \qquad \text{NS-Swap}$$

$$\left(\mu, \ell \mapsto^1 v\ ;\ \mathbf{release}^{\mathbf{w}}\ \ell\right) \longmapsto (\mu\ ;\ \mathbf{inl}\ v) \qquad \text{NS-ReleaseW1}$$

$$\left(\mu, \ell \mapsto^{i>1} v\ ;\ \mathbf{release}^{\mathbf{w}}\ \ell\right) \longmapsto \left(\mu, \ell \mapsto^{i-1} v\ ;\ \mathbf{inr}\ ()\right) \qquad \text{NS-ReleaseW2}$$

$$\left(\mu, \ell \mapsto^1 v\ ;\ \mathbf{release}^{\mathbf{s}}\ \ell\right) \longmapsto (\mu\ ;\ v) \qquad \text{NS-ReleaseS}$$

Figure 4.14: Small-Step Relation

Store Typing

$$\textsc{St-Nil}\ \frac{}{\Sigma \underset{s}{\vdash} \cdot : \cdot} \qquad \textsc{St-ConsW}\ \frac{\Sigma_1 \underset{s}{\vdash} \mu : \Sigma_2 \qquad \cdot;\cdot;\Sigma_v \vdash v : \tau}{\Sigma_1 + \Sigma_v \underset{s}{\vdash} \mu, \ell \mapsto^i v : \Sigma_2, \ell \mapsto^i_{\mathbf{w}} \tau} \qquad \textsc{St-ConsS}\ \frac{\Sigma_1 \underset{s}{\vdash} \mu : \Sigma_2 \qquad \cdot;\cdot;\Sigma_v \vdash v : \tau}{\Sigma_1 + \Sigma_v \underset{s}{\vdash} \mu, \ell \mapsto^1 v : \Sigma_2, \ell \mapsto_{\mathbf{s}} \tau}$$

Figure 4.15: Store Typing

the value we are duplicating or dropping. Every other operation that doesn't directly deal with references preserves the number of usages of locations and does not need to change the reference counts.

The reduction rules that deal explicitly with references must track the reference counts carefully. Allocating new references creates a new cell with a fresh label and a reference count of 1. The swap operation conserves reference count, but the release operations must steadily decrement the reference count of the released cell until the cell itself is deallocated and the contents extracted.

4.3.3 Store Typings

In a substructural type system, stores both consume and provide store typings. Thus the the typing relation on a store μ specifies both Σ_1 the set of typings used and Σ_2 the total set of provided bindings.

$$\boxed{\text{Store Typing: } \Sigma_1 \underset{s}{\vdash} \mu : \Sigma_2}$$

The inference rules for store typing are given in figure 4.15. Since the store does not track whether a location corresponds to a strong or weak reference, the rules are nondeterministic. The rules are also consistent with our implicit treatment of stores as sets of mappings rather than ordered lists. For instance, note that permuting the order of elements in a well-typed store yields a store which can be assigned a store typing Σ' that is merely the corresponding permutation of the old store typing. Thus, it is possible to "invert" the St-ConsW and St-ConsS rules assuming any of the labels ℓ in a store μ was the last label added.

Finally a configuration is well typed if there exists a location context under which one can both type the store and also use the remaining bindings to type the current expression. Its judgment is given in figure 4.16 on the facing page.

$$\boxed{\text{Configuration Typing: } \underset{c}{\vdash} (\mu\ ;\ e) : \tau}$$

Configurations

$$\textsc{Conf}\quad \frac{\Sigma_1 \underset{s}{\vdash} \mu : \Sigma_1 + \Sigma_2 \qquad \cdot;\cdot;\Sigma_2 \vdash e : \tau}{\underset{c}{\vdash} (\mu\ ;\ e) : \tau}$$

Figure 4.16: Configuration Typing

4.4 Type Soundness

The proof of type soundness is made complicated in λ_{cl} by the presence of references which support both strong update and deallocation. One must track the usage of variables and locations to make sure that the store remains in a consistent state.

The bulk of the work goes into proving preservation, and the key substructural lemma to prove preservation is Lemma 4.1. Intuitively, this lemma tells us that substructural constraints on a value's type respect the substructural constraints of everything the value contains / depends on.

Lemma 4.1 (Constraints Capture Locations)**.** *Consider* $P;\Gamma;\Sigma \vdash v : \tau$. *If* $P \Vdash \mathit{Dup}\ \tau$ *then* $P \Vdash \mathit{Dup}\ \Sigma, \mathit{Dup}\ \Gamma$. *Similarly if* $P \Vdash \mathit{Drop}\ \tau$ *then* $P \Vdash \mathit{Drop}\ \Sigma, \mathit{Drop}\ \Gamma$.

Proof. By induction on the typing derivation for v. See appendix A, page 67. □

Lemma 4.2 (Substitution)**.** *If* $P;\Gamma, x : \tau_x;\Sigma_1 \vdash e : \tau$ *and* $P;\cdot;\Sigma_2 \vdash v : \tau_x$ *and* $\Sigma_1 \smile \Sigma_2$ *then* $P;\Gamma;\Sigma_1 + \Sigma_2 \vdash e\,\{v/x\} : \tau$

Proof. By induction on the typing derivation for e, making use of Lemma 67 in the lambda case. See appendix A, page 68 □

In proving Preservation, it is also useful to separate out a Replacement Lemma which specifies exactly how different typing contexts must be combined together in order for one to substitute a subterm in an evaluation context.

Lemma 4.3 (Replacement)**.** *If* $P;\Gamma;\Sigma \vdash E\,[M] : \tau$ *then* $\exists \tau', \Sigma_1, \Sigma_2, \Gamma_1, \Gamma_2$ *such that*

- $\Sigma = \Sigma_1 + \Sigma_2$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$ *and* $P;\Gamma_1;\Sigma_1 \vdash M : \tau'$ *and furthermore*
- *If* $P;\Gamma_1';\Sigma_1' \vdash M' : \tau'$ *with* $\Gamma_1' \smile \Gamma_2$ *and* $\Sigma_1' \smile \Sigma_2$, *then* $P;\Gamma_1' \circ \Gamma_2;\Sigma_1' + \Sigma_2 \vdash E\,[M'] : \tau$ *for any* M', Γ_1', Σ_1'

Proof. By induction on E. See appendix A, page 70. □

Another key lemma for proving preservation relates the floc function used to maintain dynamic reference counts with the store context that a value requires. To state this lemma, we overload the floc function to also count the occurences of locations in a store context.

$$\begin{aligned}
\text{floc}\left(\overline{\ell_i \mapsto_{\mathbf{w}}^{j_i} v_i}\right) &= \underbrace{\overline{\ell_i, \cdots, \ell_i}}_{j_i} \\
\text{floc}\left(\overline{\ell_i \mapsto_{\mathbf{s}} v_i}\right) &= \overline{\ell_i} \\
\text{floc}\left(\Sigma^s, \Sigma^w\right) &= \text{floc}\left(\Sigma^s\right), \text{floc}\left(\Sigma^w\right)
\end{aligned}$$

Lemma 4.4 (Store Contexts map Free Locations)**.** *If $P; \Gamma; \Sigma \vdash e : \tau$ then floc $(e) =$ floc (Σ).*

Proof. Induction on the typing derivation. □

With Replacement and Substitution and Free Locations, proving Preservation involves mostly showing that the store changes in a way that is consistent with its store typing.

Lemma 4.5 (Preservation)**.** *If $\underset{c}{\vdash} (\mu_1 \;;\; e_1) : \tau$ and $(\mu_1 \;;\; e_1) \longmapsto (\mu_2 \;;\; e_2)$ then $\underset{c}{\vdash} (\mu_2 \;;\; e_2) : \tau$*

Proof. By casework on the single step relation. See appendix A, page 70. □

Given a standard lemma for canonical forms, the proof for Progress is straightforward and is omitted.

Lemma 4.6 (Progress)**.** *If $\underset{c}{\vdash} (\mu_1 \;;\; e_1) : \tau$ then either e_1 is a value or $\exists \mu_2, e_2$ s.t. $(\mu_1 \;;\; e_1) \longmapsto (\mu_2 \;;\; e_2)$*

With preservation and progress in hand, we can at last derive a type soundness result.

Theorem 4.1 (Type Soundness)**.** *If $\underset{c}{\vdash} (\cdot \;;\; e) : \tau$ then either it diverges or it reduces to a value configuration $(\mu \;;\; v)$ such that $\underset{c}{\vdash} (\mu \;;\; v) : \tau$.*

Proof. By Lemmas 4.5 and 4.6 and induction on the small step reduction sequence. □

Chapter 5

Language Implementation

5.1 Type Checking

To test the usability of the Clamp Type System, I implemented a type checker in Haskell which accepts expressions such as the examples given in Chapter 2 and infers a valid type scheme if the expression has one. The type checker is based off of the Haskell type checker described in Jones [1999]. As a whole, the process of extending a Haskell type-checker to support Clamp was very straightforward, and illustrates one of the strengths of the Clamp type system: it requires small and orthogonal additions upon a standard Hindley-Milner based language with type class constraints. Unless otherwise stated below, the standard parts of the type checker follow the design in Jones [1999].

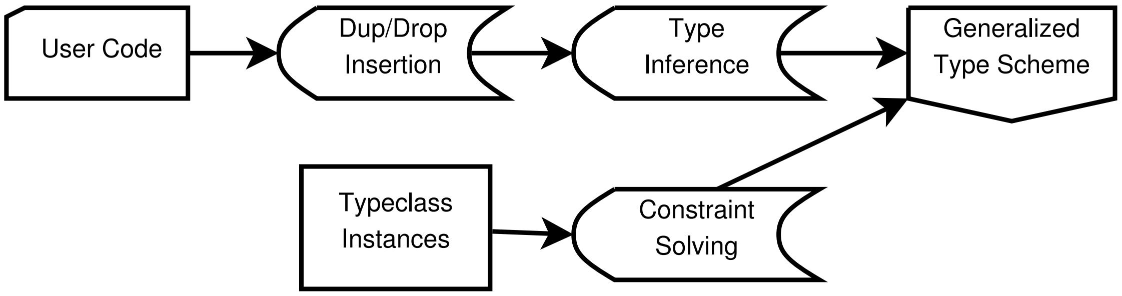

Figure 5.1: Type Checker Components

The high level structure of the type checker is given in figure 5.1. Almost all of the components are laid out just as they would be in a standard Haskell-like type-checker, with the addition of

1. A dup/drop insertion pass
2. Substructural type class instances
3. Constraints to closure environments in the type inference step

The type inference component executes a slight variant of algorithm W, performing type unification while accumulating type class constraints. The constraint solver simplifies the

Component	Lines of Code
Parser/Lexer	301
Data Structures for Expressions/Types	544
Dup/Drop Insertion	157
Type Class Operations and Instances	190
Type Inference	341
Toplevel Driver	59
Total	1592

Table 5.1: Type Checker Code Breakdown

accumulated constraints and reduces them to a normal form using instance rules which are provided separately. A generalized type scheme can then be formed from an inferred type and a canonical set of generalizable type class constraints.

In terms of the the relative sizes of each component, table 5.1 collects the number of lines of code used by each component.

The dup/drop insertion algorithm and the substructural type class instances constitute moderately sized additions to a Haskell-like type checker core, but they are self-contained components and are completely orthogonal to the general type inference and constraint solving core.

Once the dup and drop operations have been added, constraint inference can treat the dup and drop operations like any other standard primitive operation with constrained types, and given the relevant substructural type class instances the constraint solver can treat them like any other type class instances (such as those for Show or Eq).

The only significant addition to the type inference component lies in the constraints we have to add to function closures. In Clamp, unlike in Haskell, $\lambda^{aq}x.e$ imposes constraints on the environment it closes over, depending on the qualifier aq. Thus, about a dozen lines are needed in the constraint inference step to apply the relevant constraints to variables in the environment when we close over a function.

The internal kind of the arrow type constructor is also modified slightly to support the qualified arrow types $\alpha \xrightarrow{aq} \beta$ in Clamp. As described in section 2.1.6 on page 18, in the type checker we do this by treating $\rightarrow$ as a type constructor with kind $\star \rightarrow \star \rightarrow \star \rightarrow \star$ that takes an extra type argument: a dummy qualifier type.

One benefit of this architecture is that once the dup and drop operations have been inserted by a sound annotation algorithm, one does not need to track variable usages elsewhere in the type checker, since all of the substructural operations are already made explicit in the code. Outside of the dup and drop insertion step, the implemented type checker needs no knowledge of how to manipulate substructural environments, and treats all environments as unlimited just as one would in OCaml or Haskell. Type checking in a system with substructural types is thus broken down into two self-contained steps: annotating substructural usages on variables and then enforcing substructural constraints on types. The technique of separating out an explicit annotation step was used to nicely integrate a Uniqueness typing system with Hindley-Milner in Vries et al. [2008], and the technique is possibly even more effective in Clamp where the annotations are simply functions with constrained type schemes.

If we don't precisely track environments then, to apply the right set of constraints to closures our type checker can just count the free variables the lambda expression. This is sound because the insertion algorithm described in Chapter 3 guarantees that the typing environment of an annotated requires is exactly the set of free variables.

The dup/drop algorithm itself is a direct implementation of the infer_s algorithm in Chapter 3. Though the process of renaming is not described there, it is easy to implement in Haskell with a renaming state monad since if annotation is done bottom up as in infer_s, the only situation where renaming is required is when two subexpressions both use the same variable exactly once. These two usages can then be renamed to refer to the fresh names provided by a dup operation.

5.2 Memory Management

Though I have not implemented an interpreter or compiler for Clamp, the fact that Clamp includes explicit dup and drop operations allows for the integration of a reference-counting runtime system for memory management. In such a system, guarantees of linearity make it easy to avoid leaking non-cyclic data structures [Walker, 2005, Chirimar et al., 1996].

In this section we present a sketch of an abstract machine which, unlike the semantics given in Chapter 4, is aware of substructural properties and exploits them to allocate and deallocate memory efficiently. It is adapted from the ones found in Walker [2005], Chirimar et al. [1996], but unlike the abstract machine in Walker [2005] reference counts are used to manage all memory, and unlike the recursive large step semantics in Chirimar et al. [1996] this machine is formulated as a small step, stack based machine much closer to realistic implementation.

5.2.1 Refcounting Machine

It is useful in setting up this abstract machine to have it work with expressions that are in a variation of A-Normal form [Flanagan et al., 1993]. This is because A-Normal form makes is easy to identify where every allocation and deallocation occurs by flattening all of the introduction and elimination forms. Let $\lambda_{cl} - ANF$ refer to the ANF variant of λ_{cl} used in this section, and besides the A-Normal restrictions, this language otherwise shares the same type system and substructural properties as λ_{cl}.

λ_{cl}-ANF Syntax

$$\begin{aligned}
e &::= x \mid \textbf{alloc } x = I \textbf{ in } e \mid \textbf{let } x = E \textbf{ in } e \mid \textbf{letp } (x_1, x_2) = x_3 \textbf{ in } e \\
&\quad \mid \textbf{dup } x_1 \textbf{ as } y_1, y_2 \textbf{ in } e \mid \textbf{drop } x_1 \textbf{ in } e \\
E &::= x \mid x_1\ x_2 \mid \textbf{match } x \textbf{ with inl } x_1 \to e_1 ; \textbf{inr } x_2 \to e_2 \\
&\quad \mid \textbf{release}^{\textbf{rq}}\ x \mid \textbf{swap}^{\textbf{rq}}\ x_1 \textbf{ with } x_2 \\
I &::= () \mid \lambda^{aq}\,(x : \tau)\,.e \mid (x_1, x_2) \mid \textbf{inl } x \mid \textbf{inr } x \mid \textbf{new}^{\textbf{rq}}\ x
\end{aligned}$$

In the syntax presented above, introduction (I) and elimination (E) forms are distinguished to distinguish memory allocations and deallocations. Unlike most A-Normal forms, the only atomic forms are variables since the goal of this machine is to track all allocations and deallocations, even atomic ones.

Machine Components

$H ::= H, \ell \mapsto^i \langle\langle \rho, I \rangle\rangle \mid \cdot$	Heaps
$\rho ::= \rho, x \mapsto \ell \mid \cdot$	Environments
$S ::= S, \langle\langle \rho, \lambda x.e \rangle\rangle \mid \cdot$	Control Stacks

$$\boxed{\text{Machine Step: } H_1 \ ;\ S_1 \ ;\ \rho_1 \ ;\ e_1 \ \longmapsto\ H_2 \ ;\ S_2 \ ;\ \rho_2 \ ;\ e_2}$$

The abstract semantics for this machine make use of a reference-counted heap H which maps locations ℓ to closures consisting of an environment ρ and an introduction form I. Though the use of full-blown closures is not fully necessary for most introduction forms (e.g. an **inr** x form requires an environment mapping one variable to one location), it simplifies the presentation of the rules. Heaps and environments are treated as partial mappings over locations or variables, so exchange properties are assumed for both. The stack is used to manage control flow on function applications, and does not support exchange.

The intuition behind the reference counts is that the count i on a heap location $\ell \mapsto^i C$ tracks i total references to ℓ found in all of environments ρ in the heap H, in the stack S, as well as in the main environment ρ. As usual, reference counts are updated lazily, so that a pair with a reference count of two counts as a single reference to each of its components. Since all value usage is mediated by variables, all pointers into the heap are found in the environments ρ. The environments themselves use bindings linearly, so that every binding is used exactly once. Some meta-functions for manipulating reference counts are useful to define, and the ones below are inspired by the ones in Chirimar et al. [1996].

Reference Count Manipulation

$$\text{incr-ptrs } \overline{x_i \mapsto l_i} \text{ in } H, \overline{l_i \mapsto^{j_i} C_i} := H, \overline{l_i \mapsto^{j_i+1} C_i}$$

$$\text{decr-ptrs } \overline{x_i \mapsto l_i} \text{ in } H := \overline{\text{decr-ptr } \ell_i \text{ in } H}$$

$$\text{decr-ptr } \ell \text{ in } H, \ell \mapsto^{i} \langle\langle \rho, I \rangle\rangle := \begin{cases} H, \ell \mapsto^{i-1} \langle\langle \rho, I \rangle\rangle & \text{i>1} \\ \text{decr-ptrs } \rho \text{ in } H & i = 1 \end{cases}$$

$$\text{elim-clos } \ell \text{ in } H, \ell \mapsto^{i} \langle\langle \rho, I \rangle\rangle := \begin{cases} H & i = 1 \\ \text{incr-ptrs } \rho \text{ in } \left(H, \ell \mapsto^{i-1} \langle\langle \rho, I \rangle\rangle\right) & i > 1 \end{cases}$$

Intuitively, the purpose of these functions is that:

- incr-ptrs ρ in H increments the reference count on all the locations in ρ.
- decr-ptrs ρ in H decrements reference counts, but is more complicated because when a location's reference count becomes 0, its corresponding closure in the heap is deallocated. This requires all of the locations in the closure to be decremented as well.
- elim-clos ℓ in H is a function used for some elimination forms. Since some elimination forms bring a closure's attached environments into the main environment, we do not need to decrement the reference counts of locations in the attached environment, and instead need to increment their reference counts when we end up with *two* copies of ρ: one remaining in the heap and one brought into the current environment.

5.2.2 Small Step Semantics

A selection of the small step reduction rules are given below.

Since all allocation is done through creating closures on the heap, the same allocation rule can be used for all introduction forms.

Allocation Rule

$$H \ ; \ S \ ; \ \rho \ ; \ \textbf{alloc } x = I \textbf{ in } e \ \longmapsto$$

$$H, \ell \mapsto^{1} \langle\langle \rho \mid_{\text{fv}I}, I \rangle\rangle \ ; \ S \ ; \ (\rho \setminus \text{fv}I), x \mapsto \ell \ ; \ e$$

Variables are the sole atomic form in this abstract machine. To evaluate one, we pop off the next frame of the control stack and bind the argument.

Var Rule

$$H \; ; \; S, \langle\langle \rho, \lambda y.e \rangle\rangle \; ; \; \cdot, x \mapsto \ell \; ; \; x \;\longmapsto$$
$$H \; ; \; S \; ; \; \rho, y \mapsto \ell \; ; \; e$$

Duplication and dropping in this semantics update reference counts and deallocate if necessary.

Dup Rule

$$H, \ell_1 \mapsto^{i} \langle\langle \rho, I \rangle\rangle \; ; \; S \; ; \; \rho, x_1 \mapsto \ell_1 \; ; \; \textbf{dup } x_1 \textbf{ as } y_1, y_2 \textbf{ in } e \;\longmapsto$$
$$H, \ell_1 \mapsto^{i+1} \langle\langle \rho, I \rangle\rangle \; ; \; S \; ; \; \rho, y_1 \mapsto \ell_1, y_2 \mapsto \ell_1 \; ; \; e$$

Drop Rule

$$H, \ell_1 \mapsto^{i} \langle\langle \rho, I \rangle\rangle \; ; \; S \; ; \; \rho, x_1 \mapsto \ell_1 \; ; \; \textbf{drop } x_1 \textbf{ in } e \;\longmapsto$$
$$\text{decr-ptr } \ell_1 \text{ in } \big(H, \ell_1 \mapsto^{i} \langle\langle \rho, I \rangle\rangle\big) \; ; \; S \; ; \; \rho \; ; \; e$$

The elimination forms are often more complicated. For instance, when eliminating a closure in a function application, one must bring a copy of the closure environment into main memory.

Elim-App Rule

$$H, \ell_1 \mapsto^{i} \langle\langle \rho_2, \lambda y.e_1 \rangle\rangle \; ; \; S \; ; \; \rho_1, x_1 \mapsto \ell_1, x_2 \mapsto \ell_2 \; ; \; \textbf{let } x = x_1 \; x_2 \textbf{ in } e_2 \;\longmapsto$$
$$\text{elim-clos } \ell_1 \text{ in } \big(H, \ell_1 \mapsto^{i} \langle\langle \rho_2, \lambda y.e_1 \rangle\rangle\big) \; ; \; S, \langle\langle \rho_1, \lambda x.e_2 \rangle\rangle \; ; \; \rho_2, y \mapsto \ell_2 \; ; \; e_1$$

Elim-Release-Weak Rule

$$H, \ell_1 \mapsto^{1} \langle\langle y \mapsto \ell_2, \textbf{new}^{\textbf{rq}} \; y \rangle\rangle \; ; \; S \; ; \; \rho, x_1 \mapsto \ell_1 \; ; \; \textbf{let } x = \textbf{release}^{\textbf{w}} \; x_1 \textbf{ in } e$$
$$\longmapsto \; H, \ell_{ret} \mapsto^{1} \langle\langle r_1 \mapsto \ell_2, \textbf{inr } r_1 \rangle\rangle \; ; \; S \; ; \; \rho, x \mapsto \ell_{ret} \; ; \; e$$

$$H, \ell_1 \mapsto^{i} \langle\langle y \mapsto \ell_2, \textbf{new}^{\textbf{w}} \; y \rangle\rangle \; ; \; S \; ; \; \rho, x_1 \mapsto \ell_1 \; ; \; \textbf{let } x = \textbf{release}^{\textbf{w}} \; x_1 \textbf{ in } e \qquad \text{when } i > 1$$
$$\longmapsto \; H, \ell_1 \mapsto^{i-1} \langle\langle y \mapsto \ell_2, \textbf{new}^{\textbf{w}} \; y \rangle\rangle, \ell_{ret} \mapsto^{1} \langle\langle r_1 \mapsto \ell_{unit}, \textbf{inr } r_1 \rangle\rangle, \ell_{unit} \mapsto^{1} \langle\langle \cdot, () \rangle\rangle$$
$$; \; S \; ; \; \rho, x \mapsto \ell_{ret} \; ; \; e$$

The swap operation however doesn't need to modify any reference counts since it conserves its arguments.

Elim-Swap Rule

$$H, \ell_1 \mapsto^i \langle\langle y \mapsto \ell_3, \mathbf{new}\ y \rangle\rangle \ ;\ S\ ;\ \rho, x_1 \mapsto \ell_1, x_2 \mapsto \ell_2\ ;\ \mathbf{let}\ x = \mathbf{swap}\ x_1\ \mathbf{with}\ x_2\ \mathbf{in}\ e$$

$$\longmapsto\ H, \ell_1 \mapsto^i \langle\langle y \mapsto \ell_2, \mathbf{new}\ y \rangle\rangle, \ell_{ret} \mapsto^1 \langle\langle r_1 \mapsto \ell_1, r_2 \mapsto \ell_3, (r_1, r_2) \rangle\rangle\ ;\ S\ ;\ \rho, x \mapsto \ell_{ret}\ ;\ e$$

5.2.3 Reference Count Properties

The guiding principle behind the reduction rules in this reference counting machine has been to preserve two invariants.

First, variable usage is constrained by linear environments ρ, so that on valid configurations $H\ ;\ S\ ;\ \rho\ ;\ e$ we have

$$\mathrm{Dom}\,(\rho) = \mathrm{fv}\,(e)$$

where Dom is the set of variables mapped by an environment.

Second, locations in environments are tracked by reference counts in the heap, so that on a valid configuration $H\ ;\ S\ ;\ \rho\ ;\ e$ with $\ell \mapsto^i C \in H$ we have

$$i = \sum_{\langle\langle \rho_1, I_1 \rangle\rangle \in H} \mathrm{Occ}_\ell\,(\rho_1) + \sum_{\langle\langle \rho_2, I_2 \rangle\rangle \in S} \mathrm{Occ}_\ell\,(\rho_2) + \mathrm{Occ}_\ell\,(\rho)$$

where Occ_ℓ counts the number of times ℓ is mapped-to in an environment.

One can verify that the above properties are preserved by the step relation. Thus, the memory management guarantee made by this machine is that in a final value configuration $H\ ;\ \cdot\ ;\ \rho\ ;\ v$, the only closures still allocated on the heap will be those used directly in v or those which are part of cyclic structures in H. As part of a runtime system, this would allow immediate reclamation of a large number of non-cyclic data structures, and would ease the burdens put on the general garbage collector.

Chapter 6

Conclusions

6.1 Summary of Contributions

The broad direction of this thesis has been to explore how substructural types can be realized via type classes. I have focused on Clamp as one possible language that exhibits many of the benefits of this design. We can divide these benefits into roughly two categories: those which apply to language researchers and designers in studying type system theory, and those which apply to language implementors and users in putting the language to work.

- Theoretical Benefits:
  - The Clamp type system builds upon a simple and well-understood core. To System F we add only linear contexts, type class constraints, and qualified arrows.
  - The type class constraint system captures the URAL lattice succinctly, without a proliferation of kinds or type-qualifiers.
  - The separation of types and their constraints allows for natural substructural polymorphism.
- Usability Benefits:
  - Type class instances can be used to intuitively specify the behavior of state-aware data structures. For example, one can introduce a rich system of strong and weak mutable references using only two instance rules.
  - Type checking can be split into orthogonal modules. In particular, all substructural issues can be handled by dup and drop insertion pass and a set of type class instance rules, while the rest is Hindley-Milner.
  - Explicit dup and drop operations allow for a reference counting runtime system.
  - Type classes are a familiar and time-tested language feature, making programmer adoption of substructural types much less imposing.

Thus, we see that *languages based on this design can be natural for language designers, implementors, and programmers.*

Many of the ideas in Clamp are drawn together from the rich body of previous work, and I believe one of its strengths is that it relies upon very few new and ad-hoc mechanisms. However, some of the specific original contributions in this thesis include:

- The formulation of a sound type system which combines two language features not usually found together.
- The integration of a flexible system of strong and weak mutable references, along with the theoretical framework to establish their safety.
- The development of an optimal algorithm for inserting dup and drop, easing the need for programmer annotations in any language with explicit substructural operations.
- The implementation of a type checker built directly on top of existing designs.

6.2 Future Work

6.2.1 Custom duplication and dropping

The Clamp programming language relies primarily on the static properties of its type class system to support substructural types, and does not take advantage of the potential for type classes to allow custom implementations of substructural operations. In this thesis we have given possible interpretations of the dup and drop operations in terms of copying and reference counting, but these are fixed by the language design.

Allowing programmers instead to freely define their own implementations of dup and drop on custom data types would make possible usages like those seen in C++ with user defined *copy constructors* and *destructors*. One could then define data types which cleverly and implicitly managed resources like file handles whenever they were copied or deallocated. At the very least, one could define data types which managed their memory in whatever way was most appropriate, for instance by allowing either deep-copy or shallow-copy dup operations, or eager vs lazy drop operations. However, in functional languages with dup and drop inference, this kind of behavior can be unnerving since the results of simply using a variable twice become unpredictable.

6.2.2 Polymorphic Arrows

In most cases, Clamp allows programmers to define functions which are inherently polymorphic over the substructural properties of their arguments. In these cases, a function which uses one of its arguments linearly can accept any type, whether it satisfies Dup or Drop, for that argument. However, this is not the case for functions types, which are annotated with qualifiers and assigned a fixed qualifier at creation. In λ_{cl}, one cannot write a generic "compose" function that can operate over functions with different qualifiers.

Alms is able to accommodate more polymorphic arrow types by introducing a subtyping relation on qualified arrows [Tov and Pucella, 2011]. It is able to refine the types of arrows even further by introducing a language of usage qualifiers which introduces dependencies on the substructural properties of type variables. This allows one to write a function whose arrow type is annotated differently depending on what it ends up closing over.

There are many options available to increase the polymorphic expressiveness of Clamp without resorting to the complexities of subtyping. As described in section 2.1.6, in the type checker I have made qualifiers first class types, thus allowing polymorphism over qualifiers in arrow types without the need for subtyping. However, this is an ad-hoc solution, and a more principled one might incorporate ideas from qualifier based substructural languages, for instance by defining a new kind for qualifiers. The idea in Alms of expanding the language of qualifiers could also be added to Clamp. One possible adaptation of this idea would be to annotate arrow types with the types of their closure environments, in a sense assigning them a closure-converted type. The dup and drop instances for arrows could then look at the closure environment types to determine the arrow type's substructural properties. This allows more polymorphism in some cases, but exposes too much information about a function's implementation.

6.2.3 Implementation

The current implementation of the Clamp type checker showcases the core type system in λ_{cl}, and could benefit from the addition of some standard language features found in ML or Haskell. In particular, the addition of algebraic datatypes, user defined instance rules, and a module system would allow programmers to define libraries that exposed custom types with varying substructural properties.

I have also not yet implemented an interpreter or a compiler for Clamp, and it would be interesting to see if one can take advantage of reference counting systems such as the ones described in this thesis to implement an efficient compiler. Systems which reuse the memory made available by linear or affine values can perform very well [Baker, 1994].

Appendix A

Additional Proofs

A.1 Dup/Drop Insertion

Lemma (Multiset Properties)**.** *(Lemma 3.1 on page 29)*

1. $(\Gamma_1 \sqcup \Gamma_2) + (\Gamma_1 \sqcap \Gamma_2) = \Gamma_1 + \Gamma_2$

2. $(\Gamma_1 - \Gamma_2) + \Gamma_2 = \Gamma_1 \sqcup \Gamma_2$

Proof. For property (1) we use the fact that $\max(a, b) + \min(a, b) = a + b$ to simplify as follows:

$$\begin{aligned}((\Gamma_1 \sqcup \Gamma_2) + (\Gamma_1 \sqcap \Gamma_2))(x) &= (\Gamma_1 \sqcup \Gamma_2)(x) + (\Gamma_1 \sqcap \Gamma_2)(x) \\ &= \max(\Gamma_1(x), \Gamma_2(x)) + \min(\Gamma_1(x), \Gamma_2(x)) \\ &= \Gamma_1(x) + \Gamma_2(x) \\ &= (\Gamma_1 + \Gamma_2)(x)\end{aligned}$$

For property (2), note that

$$((\Gamma_1 - \Gamma_2) + \Gamma_2)(x) = (\Gamma_1(x) - \Gamma_2(x)) + \Gamma_2(x)$$

Subtraction and Addition don't associate over the nats, so we have to use casework. If $\Gamma_1(x) \leq \Gamma_2(x)$ then $\Gamma_2(x) = \max(\Gamma_1(x), \Gamma_2(x))$

$$\begin{aligned}(\Gamma_1(x) - \Gamma_2(x)) + \Gamma_2(x) &= 0 + \Gamma_2(x) \\ &= \max(\Gamma_1(x), \Gamma_2(x))\end{aligned}$$

while if $\Gamma_1(x) > \Gamma_2(x)$ then $\Gamma_1(x) = \max(\Gamma_1(x), \Gamma_2(x))$ and we can reassociate

$$\begin{aligned}(\Gamma_1(x) - \Gamma_2(x)) + \Gamma_2(x) &= \Gamma_1(x) - (\Gamma_2(x) + \Gamma_2(x)) \\ &= \Gamma_1(x) - 0 \\ &= \max(\Gamma_1(x), \Gamma_2(x))\end{aligned}$$

In either case,

$$((\Gamma_1 - \Gamma_2) + \Gamma_2)(x) = (\Gamma_1 \sqcup \Gamma_2)(x)$$

$\square$

Theorem (Inference Soundness). *(Theorem 3.1 on page 29)*
For any e, $\text{infer}_d(e)$ *is a valid derivation of* $\Gamma \vdash ae$ *for some* Γ *and* ae *where* $\text{erase}(ae) = e$.

Proof. By induction on e
Case Var: $e = x$
Trivial by the L-Var rule.
Case Lam: $e = \lambda x.e_1$
By the induction hypothesis we know that $\text{infer}_d(e_1)$ is a valid derivation of $\Gamma_1 \vdash ae_1$ where $\text{erase}(ae_1) = e_1$.

Subcase: If $x \in \Gamma_1$ then $\text{infer}_d(e) = \dfrac{\text{infer}_d(e_1) :: \Gamma_1 \vdash ae_1}{\Gamma_1 - x \vdash \lambda x.ae_1}$,
so by the L-Abs rule $\text{infer}_d(e)$ is well formed.

Subcase: If $x \notin \Gamma_1$ then $\text{infer}_d(e) = \dfrac{\dfrac{\text{infer}_d(e_1) :: \Gamma_1 \vdash ae_1}{\Gamma_1, x \vdash \textbf{drop } x \textbf{ in } ae_1}}{\Gamma_1 \vdash \lambda x.\textbf{drop } x \textbf{ in } ae_1}$,
so by the L-Abs and L-Drop rule $\text{infer}_d(e)$ is well-formed.
In either case, $\text{erase}(\text{infer}_d(e)) = \lambda x.\text{erase}(ae_1) = \lambda x.e_1$
Case Pair: $e = \langle e_1, e_2 \rangle$
By the induction hypothesis,
$\text{infer}_d(e_i)$ is a valid derivation of $\Gamma_i \vdash ae_i$ where $\text{erase}(ae_i) = e_i$.

$$\text{infer}_d(e) = \dfrac{\dfrac{\text{infer}_d(e_1) :: \Gamma_1 \vdash ae_1 \qquad \text{infer}_d(e_2) :: \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2 \rangle}}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{dup } \Gamma_1 \sqcap \Gamma_2 \textbf{ in } \langle ae_1, ae_2 \rangle}$$

The bottom inference is derivable by the L-Dup rule with $\Gamma_a = (\Gamma_1 \sqcup \Gamma_2) - (\Gamma_1 \sqcap \Gamma_2)$ and $\Gamma_b = \Gamma_1 \sqcap \Gamma_2$ since $(\Gamma_1 \sqcup \Gamma_2) + (\Gamma_1 \sqcap \Gamma_2) = \Gamma_1 + \Gamma_2$ by Lemma 3.1. Then the entire derivation is valid by the L-Pair rule.
In addition, $\text{erase}(\text{infer}_d(e)) = \text{erase}(\langle ae_1, ae_2 \rangle) = \langle e_1, e_2 \rangle$
Case With: $e = [e_1, e_2]$
By the induction hypothesis, $\text{infer}_d(e_i)$ is a valid derivation of $\Gamma_i \vdash ae_i$ where $\text{erase}(ae_i) = e_i$.

$$\text{infer}_d(e) = \cfrac{\cfrac{\text{infer}_d(e_1) :: \Gamma_1 \vdash ae_1}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{drop } \Gamma_2 - \Gamma_1 \textbf{ in } ae_1} \qquad \cfrac{\text{infer}_d(e_2) :: \Gamma_2 \vdash ae_2}{\Gamma_1 \sqcup \Gamma_2 \vdash \textbf{drop } \Gamma_1 - \Gamma_2 \textbf{ in } ae_2}}{\Gamma_1 \sqcup \Gamma_2 \vdash [\textbf{drop } \Gamma_2 - \Gamma_1 \textbf{ in } ae_1, \textbf{drop } \Gamma_1 - \Gamma_2 \textbf{ in } ae_2]}$$

The bottom inference is derivable by the L-Choice rule. The top rules are derivable by the L-Drop rules since Lemma 3.1 implies that $\Gamma_1 + (\Gamma_2 - \Gamma_1) = \Gamma_2 + (\Gamma_1 - \Gamma_2) = \Gamma_1 \sqcup \Gamma_2$.

In addition, $\text{erase}(\text{infer}_d(e)) = [\text{erase}(ae_1), \text{erase}(ae_2)] = [e_1, e_2]$. ☐

Lemma (Forced Drop)**.** *(Lemma 3.7 on page 33)*
If $\Gamma \vdash ae$ *and* $\Gamma(x) \geq 1$ *and* $x \notin fv(\text{erase}(ae))$,
then ae *contains a subterm* $\textbf{drop } \Gamma'$ **in** ae_s *where* $x \in \Gamma'$.

Proof. By induction on the derivation $\mathcal{D}$ of $\Gamma \vdash ae$

Case L-Var: $\mathcal{D} = \overline{\{x\} \vdash x}$.
Immediate since $x \in \text{fv}(x)$

Case L-Abs: $\mathcal{D} = \cfrac{\Gamma + \{y\} \vdash ae_1 \qquad y \notin \Gamma}{\Gamma \vdash \lambda y.ae_1}$
First if $\Gamma(x) \geq 1$ and $y \notin \Gamma$ then $x \neq y$.
Thus, If $x \notin \text{fv}(\text{erase}(\lambda y.ae_1))$ then $x \notin \text{fv}(\text{erase}(ae_1))$.
$(\Gamma + \{y\})(x) \geq 1$ so by the induction hypothesis ae_1 contains the appropriate drop subterm.

Case L-Pair: $\mathcal{D} = \cfrac{\Gamma_1 \vdash ae_1 \qquad \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2 \rangle}$
If $(\Gamma_1 + \Gamma_2)(x) \geq 1$ then $\Gamma_1(x) + \Gamma_2(x) \geq 1$ so either $x \in \Gamma_1$ or $x \in \Gamma_2$.
If $x \notin \text{fv}(\text{erase}(ae))$ then $x \notin \text{fv}(\text{erase}(ae_1))$ and $x \notin \text{fv}(\text{erase}(ae_2))$ so WLOG assume $x \in \Gamma_1$.
By the induction hypothesis ae_1 contains the appropriate drop subterm so we are done.

Case L-Choice: $\mathcal{D} = \cfrac{\Gamma \vdash ae_1 \qquad \Gamma \vdash ae_2}{\Gamma \vdash [ae_1, ae_2]}$
If $x \notin \text{fv}(\text{erase}(ae))$ then $x \notin \text{fv}(\text{erase}(ae_1))$ and $x \notin \text{fv}(\text{erase}(ae_2))$.
By the induction hypothesis both ae_1 and ae_2 contain the appropriate drop subterm so we are done.

Case L-Dup: $\mathcal{D} = \cfrac{\Gamma_1 + \Gamma_2 + \Gamma_2 \vdash ae_1}{\Gamma_1 + \Gamma_2 \vdash \textbf{dup } \Gamma_2 \textbf{ in } ae_1}$
$\text{erase}(ae) = \text{erase}(ae_1)$ so $x \notin \text{fv}(\text{erase}(ae_1))$.
Clearly $\Gamma \sqsubseteq \Gamma_1 + \Gamma_2 + \Gamma_2$ so $x \in \Gamma_1 + \Gamma_2 + \Gamma_2$ and by the induction hypothesis ae_1 contains the appropriate drop subterm.

Case L-Drop: $\mathcal{D} = \cfrac{\Gamma_1 \vdash ae_1}{\Gamma_1 + \Gamma_2 \vdash \textbf{drop } \Gamma_2 \textbf{ in } ae_1}$
$\text{erase}(ae) = \text{erase}(ae_1)$ so $x \notin \text{fv}(\text{erase}(ae_1))$.
If $x \in \Gamma_1 + \Gamma_2$ then either $x \in \Gamma_1$ or $x \in \Gamma_2$.

If $x \in \Gamma_1$ then by the induction hypothesis ae_1 contains the appropriate drop subterm and we are done.

If $x \in \Gamma_2$ then **drop** Γ_2 **in** ae_1 is itself the appropriate drop term and we are done. $\square$

Lemma (Forced Dup). *(Lemma 3.8 on page 33)*

If $\Gamma \vdash ae$ *and* $\Gamma(x) \leq 1$ *and there exists a subderivation* $\mathcal{D}_s :: \Gamma_s \vdash ae_s$ *of* $\Gamma \vdash ae$ *with* $\Gamma_s(x) \geq 2$,

then ae *contains a subterm* **dup** Γ' **in** ae_s *where* $x \in \Gamma'$

Proof. By induction on the derivation $\mathcal{D}$ of $\Gamma \vdash ae$

Case L-Var: $\mathcal{D} = \overline{\{x\} \vdash x}$.

Immediate since there is no appropriate subderivation $\mathcal{D}_s$.

Case L-Abs: $\mathcal{D} = \dfrac{\Gamma + \{y\} \vdash ae_1 \qquad y \notin \Gamma}{\Gamma \vdash \lambda y.ae_1}$

Note that $(\Gamma + \{y\})(x) = \Gamma(x) + \{y\}(x)$.

Either $y = x$ or $y \neq x$ but in either case since $y \notin \Gamma$, $(\Gamma + \{y\})(x) \leq 1$.

The relevant subderivation $\mathcal{D}_s$ must be inside $\Gamma + \{y\} \vdash ae_1$ so by the induction hypothesis we are done.

Case L-Pair: $\mathcal{D} = \dfrac{\Gamma_1 \vdash ae_1 \qquad \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash \langle ae_1, ae_2 \rangle}$

$\Gamma_1(x) \leq \Gamma(x)$ and $\Gamma_2(x) \leq \Gamma(x)$.

The relevant subderivation $\mathcal{D}_s$ must be inside either $\Gamma_1 \vdash ae_1$ or $\Gamma_2 \vdash ae_2$, so WLOG let it be $\Gamma_1 \vdash ae_1$.

Then by the induction hypothesis we are done.

Case L-Choice: $\mathcal{D} = \dfrac{\Gamma \vdash ae_1 \qquad \Gamma \vdash ae_2}{\Gamma \vdash [ae_1, ae_2]}$

The relevant subderivation $\mathcal{D}_s$ must be inside either $\Gamma_1 \vdash ae_1$ or $\Gamma_2 \vdash ae_2$, so WLOG let it be $\Gamma_1 \vdash ae_1$.

Then by the induction hypothesis we are done.

Case L-Dup: $\mathcal{D} = \dfrac{\Gamma_1 + \Gamma_2 + \Gamma_2 \vdash ae_1}{\Gamma_1 + \Gamma_2 \vdash \textbf{dup } \Gamma_2 \textbf{ in } ae_1}$

Consider $\Gamma(x) = (\Gamma_1 + \Gamma_2)(x)$.

If $\Gamma(x) = 0$ then $(\Gamma_1 + \Gamma_2 + \Gamma_2)(x) = 0$ so $\mathcal{D}_s$ must be inside $\Gamma_1 + \Gamma_2 + \Gamma_2 \vdash ae_1$ and by the induction hypothesis we are done.

If $\Gamma(x) = 1$ then either $\Gamma_1(x) = 1$ or $\Gamma_2(x) = 1$.

In this case if $\Gamma_1(x) = 1$ then $\Gamma_2(x) = 0$ so $(\Gamma_1 + \Gamma_2 + \Gamma_2)(x) = 1$ so by the induction hypothesis we are done.

If $\Gamma_2(x) = 1$ then the induction hypothesis does not apply but then ae itself is a subterm **dup** Γ' **in** ae_s for $\Gamma' = \Gamma_2$ and $ae_s = ae_1$.

Case L-Drop: $\mathcal{D} = \dfrac{\Gamma_1 \vdash ae_1}{\Gamma_1 + \Gamma_2 \vdash \textbf{drop } \Gamma_2 \textbf{ in } ae_1}$

If $\Gamma(x) \leq 1$ then $\Gamma_1(x) \leq 1$ and also the subderivation $\mathcal{D}_s$ must be inside ae_1 so by the induction hypothesis we are done. $\square$

A.2 λ_{cl} Type Soundness

A.2.1 Properties of Constraints and Environments

Throughout all of the proofs we will implicitly assume exchange lemmas for all of our environments and contexts. We will also often use the convention of using X_1 and X_2 to denote components of a context X where $X = X_1 + X_2$. Below we also list some of the other structural lemmas we use for reasoning about contexts and constraints.

Lemma A.1 (Composing Environment Constraints). *If $P \Vdash K(\Gamma_1)$ and $P \Vdash K(\Gamma_2)$ then $P \Vdash K(\Gamma_1 \circ \Gamma_2)$. Similarly if $P \Vdash K(\Sigma_1)$ and $P \Vdash K(\Sigma_2)$ then $P \Vdash K(\Sigma_1 + \Sigma_2)$*

Lemma A.2 (Weakening Constraints). *If $P_1; \Gamma; \Sigma \vdash e : \tau$ then $P_1 \circ P_2; \Gamma; \Sigma \vdash e : \tau$*

Lemma A.3 (Distributing Compatibility). $\Sigma_1 + \Sigma_2 \smile \Sigma_3 \iff \Sigma_1 \smile \Sigma_2 \wedge \Sigma_2 \smile \Sigma_3 \wedge \Sigma_1 \smile \Sigma_3$

Lemma A.4 (Contexts capture Free Variables). *$x \in fv(e)$ iff ($P; \Gamma; \Sigma \vdash e : \tau$ and $x \in \Gamma$).*

A.2.2 Additional Helper Lemmas

The following lemmas are stated without proof, and are useful in fleshing out some possibly omitted cases in the preservation proof.

Lemma A.5 (Type substitution in Typings). *If $P; \Gamma; \Sigma \vdash e : \tau$ and $\overline{\cdot \vdash_{wf} \tau_i}$ then*
$P\overline{\{\tau_i/\alpha_i\}}; \Gamma\overline{\{\tau_i/\alpha_i\}}; \Sigma\overline{\{\tau_i/\alpha_i\}} \vdash e\overline{\{\tau_i/\alpha_i\}} : \tau\overline{\{\tau_i/\alpha_i\}}$

Lemma A.6 (Type substitution in Constraints). *If $P \Vdash K\tau$ then $P\overline{\{\tau_i/\alpha_i\}} \Vdash K\tau\overline{\{\tau_i/\alpha_i\}}$*

A.2.3 Main Lemmas

Lemma (Constraints Capture Locations). *(Lemma 4.1 on page 49)*
Consider $P; \Gamma; \Sigma \vdash v : \tau$. If $P \Vdash Dup\ \tau$ then $P \Vdash Dup\ \Sigma, Dup\ \Gamma$. Similarly if $P \Vdash Drop\ \tau$ then $P \Vdash Drop\ \Sigma, Drop\ \Gamma$.

Note: This is not true for arbitrary e. Consider $\cdot; x : \mathbf{A}; \cdot \vdash \mathbf{new^w}\ x : \mathrm{ref}^{\mathrm{w}}\mathbf{A}$

Proof. We induct on the typing derivation $\mathcal{D}$ of $P; \Gamma; \Sigma \vdash v : \tau$

Case Cl-TAbs:

Let $\mathcal{D} = \dfrac{P \circ P_2; \Gamma; \Sigma \vdash v_1 : \tau_2 \qquad \mathrm{Dom}(P_2) \subset \overline{\alpha_i}}{P; \Gamma; \Sigma \vdash \Lambda\overline{\alpha_i}\,[P_2]\,.v_1 : \forall\overline{\alpha_i}\,[P_2]\,.\tau_2}$.

We have as an assumption that $P \Vdash \mathrm{Dup}\ \forall\overline{\alpha_i}\,[P_2]\,.\tau_2$.

By inversion on the Pred-Sch entailment rule, $P \circ P_2 \Vdash \mathrm{Dup}\ \tau_2$.

Now by induction $P \circ P_2 \Vdash \mathrm{Dup}\ \Gamma, \mathrm{Dup}\ \Sigma$.

However since $\mathrm{Dom}(P_2) \subset \overline{\alpha_i}$ for $\overline{\alpha_i}$ implicitly fresh, we know that $P \Vdash \mathrm{Dup}\ \Gamma, \mathrm{Dup}\ \Sigma$. Similarly for Drop.

Case Cl-Lam:

Let $\mathcal{D} = \dfrac{P;\Gamma,x:\tau_1;\Sigma\vdash e:\tau_2 \qquad P\Vdash \text{Constrain}^{aq}(\Gamma,\Sigma)}{P;\Gamma;\Sigma\vdash \lambda^{aq}(x:\tau_1).e:\tau_1\xrightarrow{aq}\tau_2}$.

Suppose that $P\Vdash \text{Dup }\tau_1\xrightarrow{aq}\tau_2$.

By the instance rules aq is $\mathbf{U}$ or $\mathbf{R}$.

Thus since $P\Vdash \text{Constrain}^{aq}(\Gamma,\Sigma)$ we have $P\Vdash \text{Dup }\Gamma,\text{Dup }\Sigma$. Similarly for Drop.

Case Cl-Pair/Cl-Inl/Cl-Inr:

Induction.

Case Cl-LocW and Cl-LocS:

Immediate from the definition of constraints on contexts □

Lemma (Substitution). *(Lemma 4.2 on page 49)*
If $P;\Gamma,x:\tau_x;\Sigma_1\vdash e:\tau$ *and* $P;\cdot;\Sigma_2\vdash v:\tau_x$ *and* $\Sigma_1\smile\Sigma_2$ *then* $P;\Gamma;\Sigma_1+\Sigma_2\vdash e\{v/x\}:\tau$

Note, because we need to use the fact that constraints capture contexts, this is also not true for arbitrary e in place of v.

Proof. We induct on the typing derivation $\mathcal{D}$ of $P;\Gamma,x:\tau_x;\Sigma_1\vdash e:\tau$

Case Cl-Var:

Let $\mathcal{D} = \dfrac{}{P;x:\tau;\cdot\vdash x:\tau}$.

Note that $\Gamma=\cdot$ and $\Sigma_1=\cdot$ and $\tau_x=\tau$. Thus the conclusion is trivial.

Case Cl-Tabs:

Let $\mathcal{D} = \dfrac{P,P_2;\Gamma,x:\tau_x;\Sigma_1\vdash v_1:\tau_1 \qquad \text{Dom}(P_2)\subset\overline{\alpha_i}}{P;\Gamma,x:\tau_x;\Sigma_1\vdash \Lambda\overline{\alpha_i}[P_2].v_1:\forall\overline{\alpha_i}[P_2].\tau_1}$

By the constraint weakening lemma, $P,P_2;\cdot;\Sigma_2\vdash v:\tau_x$.

Thus by the induction hypothesis, $P,P_2;\Gamma;\Sigma_1+\Sigma_2\vdash v_1\{v/x\}:\tau_1$.

Then by Cl-Tabs we are done.

Case Cl-Tapp:

Let $\mathcal{D} = \dfrac{P;\Gamma,x:\tau_x;\Sigma_1\vdash e:\forall\overline{\alpha_i}[P_2].\tau_1 \qquad P\Vdash P_2\overline{\{\tau_i'/\alpha_i\}}}{P;\Gamma,x:\tau_x;\Sigma_1\vdash e\left[\overline{\tau_i'}\right]:\tau_1\overline{\{\tau_i'/\alpha_i\}}}$

By Induction.

Case Cl-Lam:

Let $\mathcal{D} = \dfrac{P;\Gamma,x:\tau_x,y:\tau_1;\Sigma_1\vdash e_1:\tau_2 \qquad P\Vdash \text{Constrain}^{aq}(\Gamma,x:\tau_x,\Sigma_1)}{P;\Gamma,x:\tau_x;\Sigma_1\vdash \lambda^{aq}(y:\tau_1).e_1:\tau_1\xrightarrow{aq}\tau_2}$.

By induction $P;\Gamma,y:\tau_1;\Sigma_1+\Sigma_2\vdash e_1\{v/x\}:\tau_2$.

If $aq=\mathbf{L}$ then by Cl-Lam we are done.

Consider if $aq=\mathbf{U}$.

Since $P\Vdash \text{Dup }\tau_x,\text{Drop }\tau_x$, by Lemma 4.1,

$P\Vdash \text{Dup }\Sigma_2,\text{Drop }\Sigma_2$, so $P\Vdash \text{Constrain}^{\mathbf{U}}(\Gamma,\Sigma_1+\Sigma_2)$.

Then using Cl-Lam we are done. The cases for **R** and **L** are analogous.

Case Cl-App:

Let $\mathcal{D} = \dfrac{P;\Gamma_1,x:\tau_x;\Sigma_{11}\vdash e_1:\tau_2\xrightarrow{aq}\tau \qquad P;\Gamma_2;\Sigma_{12}\vdash e_2:\tau_2}{P;\Gamma_1\circ\Gamma_2,x:\tau_x;\Sigma_{11}+\Sigma_{12}\vdash e_1\ e_2:\tau}$

In this case since e_1 is typed with x in its context, $x\in\text{fv}(e_1)$, so $e\{v/x\}=e_1\{v/x\}\,e_2$.

The other case where $P;\Gamma_2,x:\tau_x;\Sigma_{12}\vdash e_2:\tau_2$ and $e\{v/x\}=e_1e_2\{v/x\}$is analogous.

$\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Sigma_1 = \Sigma_{11} + \Sigma_{12}$.
Since $(\Sigma_{11} + \Sigma_{12}) \smile \Sigma_2$, $\Sigma_{11} \smile \Sigma_2$ and $\Sigma_{12} \smile \Sigma_2$.
Then by induction: $P; \Gamma_1, x : \tau_x; \Sigma_{11} + \Sigma_2 \vdash e_1 \{v/x\} : \tau_2 \xrightarrow{aq} \tau$.
We know from Cl-App that $\Sigma_{11} \smile \Sigma_{12}$, so we can derive that $\Sigma_{11} + \Sigma_2 \smile \Sigma_{12}$.
Now we can apply Cl-App to finish.
Case Cl-Dup:
Subcase 1:

$$\text{Let } \mathcal{D} = \frac{P; \Gamma_1, x : \tau_x; \Sigma_{11} \vdash e_1 : \tau_1 \qquad P; \Gamma_2, x_1 : \tau_1, x_2 : \tau_1; \Sigma_{12} \vdash e_2 : \tau \qquad P \Vdash \text{Dup } \tau_1}{P; \Gamma_1 \circ \Gamma_2, x : \tau_x; \Sigma_{11} + \Sigma_{12} \vdash \textbf{dup } e_1 \textbf{ as } x_1, x_2 \textbf{ in } e_2 : \tau}$$

In this subcase, $e\{v/x\} = \textbf{dup } e_1 \{v/x\} \textbf{ as } x_1, x_2 \textbf{ in } e_2$
By compatibility arguments, $\Sigma_{11} \smile \Sigma_2$ so by induction $P; \Gamma_1; \Sigma_{11} + \Sigma_2 \vdash e_1 \{v/x\} : \tau_1$.
By compatibility arguments, $\Sigma_{11} + \Sigma_2 \smile \Sigma_{12}$.
Then we can apply Cl-Dup and we are done.
The other subcase is analogous
Case Cl-Drop:
Analogous to Cl-Dup
Case Cl-Pair:
Analogous to Cl-App
Case Cl-Inl/Cl-Inr:
Induction
Case Cl-Letp:
Analogous to Cl-App
Case Cl-Match:
Subcase 1:

$$\text{Let } \mathcal{D} = \frac{\begin{array}{c} P; \Gamma_1, x : \tau_x; \Sigma_{11} \vdash e_1 : \tau_{11} + \tau_{12} \\ P; \Gamma_2, x_{21} : \tau_{11}; \Sigma_{12} \vdash e_{21} : \tau \qquad P; \Gamma_2, x_{22} : \tau_{12}; \Sigma_{12} \vdash e_{22} : \tau \end{array}}{P; \Gamma_1 \circ \Gamma_2, x : \tau_x; \Sigma_{11} + \Sigma_{12} \vdash \textbf{match } e_1 \textbf{ with inl } x_{21} \to e_{21}; \textbf{inr } x_{22} \to e_{22} : \tau}$$

In this subcase $e\{v/x\} = \textbf{match } e_1 \{v/x\} \textbf{ with inl } x_{21} \to e_{21}; \textbf{inr } x_{22} \to e_{22}$.
Thus we can apply the induction hypothesis to $P; \Gamma_1, x : \tau_x; \Sigma_{11} \vdash e_1 : \tau_{11} + \tau_{12}$ and then use compatibility arguments to use Cl-Match to finish.
Subcase 2:

$$\mathcal{D} = \frac{\begin{array}{c} P; \Gamma_1; \Sigma_{11} \vdash e_1 : \tau_{11} + \tau_{12} \\ P; \Gamma_2, x_{21} : \tau_{11}, x : \tau_x; \Sigma_{12} \vdash e_{21} : \tau \qquad P; \Gamma_2, x_{22} : \tau_{12}, x : \tau_x; \Sigma_{12} \vdash e_{22} : \tau \end{array}}{P; \Gamma_1 \circ \Gamma_2, x : \tau_x; \Sigma_{11} + \Sigma_{12} \vdash \textbf{match } e_1 \textbf{ with inl } x_{21} \to e_{21}; \textbf{inr } x_{22} \to e_{22} : \tau}$$

In this subcase $e\{v/x\} = \textbf{match } e_1 \textbf{ with inl } x_{21} \to e_{21} \{v/x\}; \textbf{inr } x_{22} \to e_{22} \{v/x\}$.
Since $\Sigma_{11} + \Sigma_{12} \smile \Sigma_2$ then $\Sigma_{12} \smile \Sigma_2$.
Thus we can apply the induction hypothesis twice to get that:
$P; \Gamma_2, x_{21} : \tau_{11}; \Sigma_{12} + \Sigma_2 \vdash e_{21} \{v/x\} : \tau$ and $P; \Gamma_2, x_{22} : \tau_{12}; \Sigma_{12} + \Sigma_2 \vdash e_{22} \{v/x\} : \tau$.
By compatibility arguments, $\Sigma_{12} + \Sigma_2 \smile \Sigma_{11}$, so we can apply Cl-Match to finish.
Case Cl-LocW/ LocS:
Trivial
Case Cl-New/Cl-ReleaseS/W:
Induction
Case Cl-SwapS/W:

Analogous to CL-App ☐

Lemma (Replacement). *(Lemma 4.3 on page 49)*
If $P;\Gamma;\Sigma \vdash E\,[M] : \tau$ *then* $\exists \tau', \Sigma_1, \Sigma_2, \Gamma_1, \Gamma_2$ *such that*

- $\Sigma = \Sigma_1 + \Sigma_2$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$ *and* $P;\Gamma_1;\Sigma_1 \vdash M : \tau'$ *and furthermore*
- *If* $P;\Gamma_1^{'};\Sigma_1^{'} \vdash M' : \tau'$ *with* $\Gamma_1^{'} \smile \Gamma_2$ *and* $\Sigma_1^{'} \smile \Sigma_2$, *then* $P;\Gamma_1^{'} \circ \Gamma_2;\Sigma_1^{'} + \Sigma_2 \vdash E\,[M'] : \tau$

Proof. By induction on E.
From our hypotheses we have that $P;\Gamma;\Sigma \vdash E\,[M] : \tau$.
Case $E = [\cdot]$:
Trivially with $\Sigma = \Sigma_1$, $\Gamma = \Gamma_1$, $\tau' = \tau$
Case $E = E_1\ e$**:**
By inverting the Cl-App rule $P;\Gamma_a;\Sigma_a \vdash E_1\,[M] : \tau_1 \xrightarrow{aq} \tau$ and $P;\Gamma_b;\Sigma_b \vdash e : \tau_1$.
Now by the induction hypothesis, $\exists \Gamma_{a1}, \Gamma_{a2}, \Sigma_{a1}, \Sigma_{a2}, \tau'$ such that $\Gamma_a = \Gamma_{a1} \circ \Gamma_{a2}$, $\Sigma_b = \Sigma_{a1} + \Sigma_{a2}$, $P;\Gamma_{a1};\Sigma_{a1} \vdash M : \tau'$
and if $P;\Gamma_{a1}^{'};\Sigma_{a1}^{'} \vdash M' : \tau'$ with $\Gamma_{a1}^{'} \smile \Gamma_{a2}$ and $\Sigma_{a1}^{'} \smile \Sigma_{a2}$, then $P;\Gamma_{a1}^{'} \circ \Gamma_{a2};\Sigma_{a1}^{'} + \Sigma_{a2} \vdash E_1\,[M'] : \tau_1 \xrightarrow{aq} \tau$.
Let $\Gamma_1 = \Gamma_{a1}, \Sigma_1 = \Sigma_{a1}, \Gamma_2 = \Gamma_{a2} \circ \Gamma_b, \Sigma_b = \Sigma_{a2} + \Sigma_b$ so that $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Sigma = \Sigma_1 + \Sigma_2$ as desired.
The consistency of these context joins is guaranteed from inverting the Cl-App rule and by the induction hypothesis.
If $P;\Gamma_1^{'};\Sigma_1^{'} \vdash M' : \tau'$ with $\Gamma_1^{'} \smile \Gamma_2$ and $\Sigma_1^{'} \smile \Sigma_2$ then $\Gamma_1^{'} \smile \Gamma_{a2}$ and $\Sigma_1^{'} \smile \Sigma_{a2}$ so $P;\Gamma_1^{'} \circ \Gamma_{a2};\Sigma_1^{'} + \Sigma_{a2} \vdash E_1\,[M'] : \tau_1 \xrightarrow{aq} \tau$.
Now by the Cl-App rule, we can check all of the context compatibility contexts again and we are done.
Case Remaining: Either analogous to the $E = E_1\ e$ case or by straightforward induction. ☐

Lemma (Preservation). *(Lemma 4.5 on page 50)*
If $\underset{c}{\vdash} (\mu_1\ ;\ e_1) : \tau$ *and* $(\mu_1\ ;\ e_1) \longmapsto (\mu_2\ ;\ e_2)$ *then* $\underset{c}{\vdash} (\mu_2\ ;\ e_2) : \tau$

Proof. Consider the one step relation. We can transform any derivation of the step relation into one involving a single use of the NS-Context rule at the root, and then another rule R above that.
Thus we have $e_1 = E\left[e_1^{'}\right]$, $e_2 = E\left[e_2^{'}\right]$ and $\left(\mu_1\ ;\ E\left[e_1^{'}\right]\right) \longmapsto \left(\mu_2\ ;\ E\left[e_2^{'}\right]\right)$ with $\Sigma_1 \underset{s}{\vdash} \mu_1 : \Sigma_1 + \Sigma_2$ and $\cdot;\cdot;\Sigma_2 \vdash E\left[e_1^{'}\right] : \tau$ from inverting $\underset{c}{\vdash} (\mu_1\ ;\ e_1) : \tau$.
By the replacement lemma, $\exists \tau', \Sigma_{21}, \Sigma_{22}$ such that $\Sigma_2 = \Sigma_{21} + \Sigma_{22}$ and

$$\cdot;\cdot;\Sigma_{21} \vdash e_1^{'} : \tau'$$

so that when $\cdot;\cdot;\Sigma_{21b} \vdash e_{1b}^{'} : \tau'$ with $\Sigma_{21b} \smile \Sigma_{22}$ then $\cdot;\cdot;\Sigma_{21b} + \Sigma_{22} \vdash E\left[e_{1b}^{'}\right] : \tau$

To complete the proof, we need to consider all the possible cases for the rule

$$R :: \left(\mu_1\ ;\ e_1^{'}\right) \longmapsto \left(\mu_2\ ;\ e_2^{'}\right)$$

The majority of these are are straightforward but long applications of the replacement lemma and manipulations of store typings. We work through three representative cases below.

Case NS-BetaV:

In this case, $R :: \left(\mu_1 \ ; \ \lambda\,(x:\tau)\,.e_{11}^{'}\ v_{12}^{'}\right) \ \longmapsto\ \left(\mu_1 \ ; \ e_{11}^{'}\left\{v_{12}^{'}/x\right\}\right)$

By inversion on $\cdot;\cdot;\Sigma_{21} \vdash e_1^{'} : \tau'$ and the substitution lemma we know that $\cdot;\cdot;\Sigma_{21} \vdash e_{11}^{'}\left\{v_{12}^{'}/x\right\} : \tau'$ or in other words $\cdot;\cdot;\Sigma_{21} \vdash e_2^{'} : \tau'$.

Thus by the replacement lemma, $\cdot;\cdot;\Sigma_2 \vdash E\left[e_2^{'}\right] : \tau$ and we are done since μ is unchanged.

Case NS-Swap:

Here we have:

$R :: \left(\mu_{11}, \ell \mapsto^i v_1 \ ; \ \mathbf{swap^{rq}}\ \ell\ \mathbf{with}\ v_{12}^{'}\right) \ \longmapsto\ \left(\mu_{11}, \ell \mapsto^i v_{12}^{'} \ ; \ (\ell, v_1)\right)$

Thus $\mu_1 = \mu_{11}, \ell \mapsto^i v_1$, $e_1^{'} = \mathbf{swap^{rq}}\ \ell\ \mathbf{with}\ v_{12}^{'}$

Consider the derivation of $\cdot;\cdot;\Sigma_{21} \vdash \mathbf{swap^{rq}}\ \ell\ \mathbf{with}\ v_{12}^{'} : \tau'$.

There are two ways to invert this derivation

Subcase Cl-SwapW:

If we invert using the Cl-SwapW rule, we get

$\tau' = \mathrm{ref}^{\mathrm{rq}}\tau_1^{'} \times \tau_1^{'}$ and $\Sigma_{21} = \Sigma_{211}, \ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'}$ with $\cdot;\cdot;\ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'} \vdash \ell : \mathrm{ref}^{\mathrm{rq}}\tau_1^{'}$ and $\cdot;\cdot;\Sigma_{211} \vdash v_{12}^{'} : \tau_1^{'}$.

By inversion on the St-ConsW or St-ConsS rule on $\Sigma_1 \underset{s}{\vdash} \mu_{11}, \ell \mapsto^i v_1 : \Sigma_1 + \Sigma_2$ since $\ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'} \in \Sigma_1 + \Sigma_2$,

we have $\Sigma_{11} \underset{s}{\vdash} \mu_{11} : \Sigma_1 + \Sigma_2 - \left\{\ell \mapsto_{\mathbf{rq}}^{i} \tau_1^{'}\right\}$ and $\Sigma_{12} \underset{s}{\vdash} v_1 : \tau_1^{'}$ where $\Sigma_{11} + \Sigma_{12} = \Sigma_1$.

By the Cl-Pair rule, $\cdot;\cdot;\ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'}, \Sigma_{12} \vdash (\ell, v_1) : \mathrm{ref}^{\mathrm{rq}}\tau_1^{'} \times \tau_1^{'}$ and since $\Sigma_1 \smile \Sigma_2$ and $\Sigma_{21} \smile \Sigma_{22}$ we have that $\left(\ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'}, \Sigma_{12}\right) \smile \Sigma_{22}$.

Thus by the replacement lemma on $e_2^{'} = (\ell, v_1)$, we have that $\cdot;\cdot;\ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'}, \Sigma_{12} + \Sigma_{22} \vdash E\left[e_2^{'}\right] : \tau$.

$\Sigma_{11} \smile \Sigma_{211}$ since $\Sigma_1 \smile \Sigma_2$, so by the St-ConsW or St-ConsS rule we can also type the new store μ_2 as $\Sigma_{11} + \Sigma_{211} \underset{s}{\vdash} \mu_{11}, \ell \mapsto^i v_{12}^{'} : \Sigma_1 + \Sigma_2 - \left\{\ell \mapsto_{\mathbf{rq}}^{i} \tau_1^{'}\right\} + \left\{\ell \mapsto_{\mathbf{rq}}^{i} \tau_1^{'}\right\}$

Finally, since

$\left(\ell \mapsto_{\mathbf{rq}}^{1} \tau_1^{'}, \Sigma_{12} + \Sigma_{22}\right) + (\Sigma_{11} + \Sigma_{211}) = \Sigma_1 + \Sigma_2 - \left\{\ell \mapsto_{\mathbf{rq}}^{i} \tau_1^{'}\right\} + \left\{\ell \mapsto_{\mathbf{rq}}^{i} \tau_1^{'}\right\}$

by the Conf rule we can conclude that $\underset{c}{\vdash} \left(\mu_{11}, \ell \mapsto^i v_{12}^{'} \ ; \ E\left[e_2^{'}\right]\right) : \tau$

Subcase Cl-SwapS:

If we invert using the Cl-SwapS rule, we get

$\tau' = \mathrm{ref}^{\mathrm{s}}\tau_2^{'} \times \tau_1^{'}$ and $\Sigma_{21} = \Sigma_{211}, \ell \mapsto_{\mathbf{s}} \tau_1^{'}$ with $\cdot;\cdot;\ell \mapsto_{\mathbf{s}} \tau_1^{'} \vdash \ell : \mathrm{ref}^{\mathrm{s}}\tau_1^{'}$ and $\cdot;\cdot;\Sigma_{211} \vdash v_{12}^{'} : \tau_2^{'}$.

By inversion on the St-ConsS rule on $\Sigma_1 \underset{s}{\vdash} \mu_{11}, \ell \mapsto^1 v_1 : \Sigma_1 + \Sigma_{22} + \left(\Sigma_{211}, \ell \mapsto_{\mathbf{s}} \tau_1^{'}\right)$ since strong references can only occur once in a well formed store context,

we have $\Sigma_{11} \underset{s}{\vdash} \mu_{11} : \Sigma_1 + \Sigma_{22} + \Sigma_{211}$ and $\Sigma_{12} \underset{s}{\vdash} v_1 : \tau_1^{'}$ where $\Sigma_{11} + \Sigma_{12} = \Sigma_1$.

By the Cl-Pair rule, $\cdot;\cdot;\ell \mapsto_{\mathbf{s}} \tau_2^{'}, \Sigma_{12} \vdash (\ell, v_1) : \mathrm{ref}^{\mathrm{s}}\tau_2^{'} \times \tau_1^{'}$ and since $\Sigma_1 \smile \Sigma_2$ and $\Sigma_{21} \smile \Sigma_{22}$ and $\ell \in \Sigma_{21}$ but $\ell \notin \Sigma_{22}$, we have that $\left(\ell \mapsto_{\mathbf{s}} \tau_2^{'}, \Sigma_{12}\right) \smile \Sigma_{22}$.

Thus by the replacement lemma on $e_2^{'} = (\ell, v_1)$, we have that $\cdot;\cdot;\ell \mapsto_{\mathbf{s}} \tau_2^{'}, \Sigma_{12} + \Sigma_{22} \vdash E\left[e_2^{'}\right] : \tau$.

$\Sigma_{11} \smile \Sigma_{211}$ since $\Sigma_1 \smile \Sigma_2$, so by the St-ConsS rule we can also type the new store μ_2 as $\Sigma_{11} + \Sigma_{211} \underset{s}{\vdash} \mu_{11}, \ell \mapsto^1 v_{12}^{'} : \Sigma_1 + \Sigma_{22} + \Sigma_{211} + \left\{\ell \mapsto_{\mathbf{s}} \tau_2^{'}\right\}$

Finally, since
$\left(\ell \mapsto_{\mathbf{s}} \tau_2', \Sigma_{12} + \Sigma_{22}\right) + (\Sigma_{11} + \Sigma_{211}) = \Sigma_1 + \Sigma_{22} + \Sigma_{211} + \left\{\ell \mapsto_{\mathbf{s}} \tau_2'\right\}$
by the Conf rule we can conclude that $\underset{c}{\vdash} \left(\mu_{11}, \ell \mapsto^1 v_{12}' \ ; \ E\left[e_2'\right]\right) : \tau$

Case NS-Dup:

In this case:
$R :: \left(\mu_1 \ ; \ \mathbf{dup}\ v_{11}' \ \mathbf{as}\ x_1, x_2 \ \mathbf{in}\ e_{12}'\right) \longmapsto \left(\text{incr floc}\left(v_{11}'\right) \text{ in } \mu_1 \ ; \ e_{12}' \left\{v_{11}'/x_1\right\}\left\{v_{11}'/x_2\right\}\right)$
Consider the derivation of $\cdot;\cdot;\Sigma_{21} \vdash \mathbf{dup}\ v_{11}' \ \mathbf{as}\ x_1, x_2 \ \mathbf{in}\ e_{12}' : \tau'$.
By inversion, $\cdot;\cdot;\Sigma_{211} \vdash v_{11}' : \tau_1'$ and $\cdot; x_1 : \tau_1', x_2 : \tau_1'; \Sigma_{212} \vdash e_{12}' : \tau'$ where $\Sigma_{21} = \Sigma_{211} + \Sigma_{212}$ and $\cdot \Vdash \text{Dup } \tau_1'$.
Since $\Sigma_{21} = \Sigma_{211} + \Sigma_{212}$ we know $\Sigma_{211} \smile \Sigma_{212}$ so by the substitution lemma, $\cdot; x_2 : \tau_1'; \Sigma_{211} + \Sigma_{212} \vdash e_{12}' \left\{v_{11}'/x_1\right\} : \tau'$.
By the Constraint Captures Locations lemma (Lemma 4.1) since $\cdot \Vdash \text{Dup } \tau_1'$ then Dup Σ_{211}.
This means Σ_{211} can only contain weak location bindings so $\Sigma_{211} \smile \Sigma_{211}$.
Then, by the substitution lemma again, $\cdot;\cdot;\Sigma_{211} + \Sigma_{212} + \Sigma_{211} \vdash e_{12}' \left\{v_{11}'/x_1\right\}\left\{v_{11}'/x_2\right\} : \tau'$
The relation $\Sigma_{21} + \Sigma_{211} \smile \Sigma_{22}$ holds so by the replacement lemma $\cdot;\cdot;\Sigma_2 + \Sigma_{211} \vdash E\left[e_{12}' \left\{v_{11}'/x_1\right\}\left\{v_{11}'/x_2\right\}\right] : \tau$
Remember that $\Sigma_1 \underset{s}{\vdash} \mu_1 : \Sigma_1 + \Sigma_2$.
By the Free Locations lemma (Lemma 4.4) $\text{floc}\left(v_{11}'\right) = \text{floc}\left(\Sigma_{211}\right)$.
Again, Σ_{211} can contain only weak location bindings so $\Sigma_{211} \smile \Sigma_2$ and $\Sigma_1 \underset{s}{\vdash} \text{incr floc}\left(v_{11}'\right) \text{ in } \mu_1 : \Sigma_1 + \Sigma_2 + \Sigma_{211}$.
Finally, by the Conf rule we have
$\underset{c}{\vdash} \left(\text{incr floc}\left(v_{11}'\right) \text{ in } \mu_1 \ ; \ E\left[e_{12}' \left\{v_{11}'/x_1\right\}\left\{v_{11}'/x_2\right\}\right]\right) : \tau$ □

Bibliography

S. Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1-2):3–57, Apr. 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90181-R. URL `http://dx.doi.org.ezp-prod1.hul.harvard.edu/10.1016/0304-3975(93)90181-R`.

A. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 78–91, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086376. URL `http://doi.acm.org/10.1145/1086365.1086376`.

H. G. Baker. A "linear logic" quicksort. *SIGPLAN Not.*, 29(2):13–18, Feb. 1994. ISSN 0362-1340. doi: 10.1145/181748.181750. URL `http://doi.acm.org/10.1145/181748.181750`.

H. G. Baker. "use-once" variables and linear objects: storage management, reflection and multi-threading. *SIGPLAN Not.*, 30(1):45–52, Jan. 1995. ISSN 0362-1340. doi: 10.1145/199818.199860. URL `http://doi.acm.org/10.1145/199818.199860`.

T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503274. URL `http://doi.acm.org/10.1145/503272.503274`.

E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, pages 579–612, 1996.

G. M. Bierman. *On Intuitionistic Linear Logic*. PhD thesis, University of Cambridge, 1993.

J. Cheney and G. Morrisett. A linearly typed assembly language. Technical report, Cornell University, Ithaca, NY, USA, February 2003.

J. Chirimar, C. A. Gunter, and J. G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6:6–2, 1996.

K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292564. URL `http://doi.acm.org/10.1145/292540.292564`.

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 59–69, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378811. URL `http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/378795.378811`.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155113. URL `http://doi.acm.org/10.1145/155090.155113`.

J. Y. Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7, 1972.

J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, Jan. 1987. ISSN 0304-3975. doi: 10.1016/0304-3975(87)90045-4. URL `http://dx.doi.org.ezp-prod1.hul.harvard.edu/10.1016/0304-3975(87)90045-4`.

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199475. URL `http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/199448.199475`.

M. P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-47253-9.

M. P. Jones. Typing haskell in haskell. In *Haskell Workshop*, 1999.

K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in system f. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10, pages 77–88, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-891-9. doi: 10.1145/1708016.1708027. URL `http://doi.acm.org/10.1145/1708016.1708027`.

M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224195. URL `http://doi.acm.org/10.1145/224164.224195`.

F. Pottier and D. Remy. *Advanced Types and Programming Languages*, chapter The Essence of ML Type Inference. MIT Press, 2005.

G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Sci. Comput. Program.*, 23(2-3):197–226, Dec. 1994. ISSN 0167-6423. doi: 10.1016/0167-6423(94)00020-4. URL `http://dx.doi.org/10.1016/0167-6423(94)00020-4`.

P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269, Nov. 2005. ISSN 0164-0925. doi: 10.1145/1108970.1108974. URL `http://doi.acm.org/10.1145/1108970.1108974`.

N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in cyclone. *Sci. Comput. Program.*, 62(2):122–144, Oct. 2006. ISSN 0167-6423. doi: 10.1016/j.scico.2006.02.003. URL `http://dx.doi.org/10.1016/j.scico.2006.02.003`.

J. A. Tov and R. Pucella. Practical affine types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926436. URL `http://doi.acm.org/10.1145/1926385.1926436`.

D. N. Turner and P. Wadler. Operational interpretations of linear logic. *Theor. Comput. Sci.*, 227(1-2):231–248, Sept. 1999. ISSN 0304-3975. doi: 10.1016/S0304-3975(99)00054-7. URL `http://dx.doi.org/10.1016/S0304-3975(99)00054-7`.

D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 1–11, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224168. URL `http://doi.acm.org/10.1145/224164.224168`.

E. Vries, R. Plasmeijer, and D. M. Abrahamson. Implementation and application of functional languages. chapter Uniqueness Typing Simplified, pages 201–218. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85372-5. doi: 10.1007/978-3-540-85373-2_12. URL `http://dx.doi.org.ezp-prod1.hul.harvard.edu/10.1007/978-3-540-85373-2_12`.

P. Wadler. Is there a use for linear logic? In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '91, pages 255–273, New York, NY, USA, 1991. ACM. ISBN 0-89791-433-3. doi: 10.1145/115865.115894. URL `http://doi.acm.org/10.1145/115865.115894`.

P. Wadler. A taste of linear logic (invited talk). In *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, Gdansk, Poland, Aug. 1993. Springer Verlag.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283. URL `http://doi.acm.org/10.1145/75277.75283`.

D. Walker. *Advanced Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.