# Type Classes for Lightweight Substructural Types

Edward Gan
Facebook, Menlo Park
edgan8@gmail.com

Jesse A. Tov
Northeastern University, Boston
tov@ccs.neu.edu

Greg Morrisett
Harvard University, Cambridge
greg@eecs.harvard.edu

Linear and substructural types are powerful tools, but adding them to standard functional programming languages often means introducing extra annotations and typing machinery. We propose a lightweight substructural type system design that recasts the structural rules of weakening and contraction as type classes; we demonstrate this design in a prototype language, Clamp.

Clamp supports polymorphic substructural types as well as an expressive system of mutable references. At the same time, it adds little additional overhead to a standard Damas–Hindley–Milner type system enriched with type classes. We have established type safety for the core model and implemented a type checker with type inference in Haskell.

## 1 Introduction

Type classes [12, 19] provide a way to constrain types by the operations they support. If the type class predicate $\mathsf{Dup}\,\alpha$ indicates when assumptions of type $\alpha$ are subject to contraction (duplication), and $\mathsf{Drop}\,\alpha$ indicates whether they are subject to weakening (dropping), then linear, relevant, affine, and unlimited typing disciplines are all enforced by some subset of these classes. Linear types, then, are types that satisfy neither Dup nor Drop. This idea, suggested in one author's dissertation [16], forms the basis of our prototype substructural programming language Clamp.

Clamp programs are written in a Haskell-like external language in which weakening and contraction are implicit. This is easier for programmers to work with, but to specify the type system and semantics the external language is elaborated into an internal language that is linear (i.e. variables are used exactly once.) The internal language provides explicit *dup* and *drop* operations, which impose the corresponding type class constraints on their arguments. Thus, in the internal language one might think of *dup* and *drop* as functions with these qualified types:

$$\mathit{dup} : \forall\alpha.\,\mathsf{Dup}\,\alpha \Rightarrow \alpha \to \alpha \times \alpha$$
$$\mathit{drop} : \forall\alpha\beta.\,\mathsf{Drop}\,\alpha \Rightarrow \alpha \to \beta \to \beta$$

In the internal language all nonlinear usage is mediated by the dup and drop operations. For example, the internal language term $\lambda x.\,x+x$ is ill formed because it uses variable $x$ twice, but the term

$$\lambda x.\,\mathbf{let}\ (x_1,x_2) = \mathit{dup}\ x\ \mathbf{in}\ x_1+x_2$$

is well typed. Because elaboration into the internal language ensures that the resulting program is linear, it can then be checked using nearly-standard Damas–Hindley–Milner type reconstruction [8] with type classes [12, 19]; improper duplication and dropping is indicated by unsatisfiable type class constraints.

S. Alves and I. Cervesato (Eds): Third International Workshop
on Linearity 2014 (LINEARITY'14)
EPTCS 176, 2015, pp. 34–48, doi:10.4204/EPTCS.176.4

```
fst    :: Drop b => (a, b) -U> a
fst     = \(x, y) -U> x

constU :: (Dup a, Drop a, Drop b) => a -U> b -U> a
constU  = \x -U> \y -U> x

constL :: Drop b => a -U> b -L> a
constL  = \x -U> \y -L> x
```

Figure 1: Prelude functions with inferred signatures

**Contributions.** We believe that Clamp offers substructural types with less fuss than many prior approaches to programmer-facing substructural type systems. Throughout the design, we leverage standard type class machinery to deal with most of the constraints imposed by substructural types. Implementing type inference for Clamp (§3) is straightforward and it is also easy to extend the system with custom resource aware structures (§2.4). The specific contributions in this paper include:

- a type system design with polymorphic substructural types and a type safety theorem (§2);
- a flexible system for managing weak and strong references (§2.4);
- a type checker with type inference derived from a type checker for Haskell (§3); and
- a dup-and-drop–insertion algorithm that is in some sense optimal (§3.1).

## 1.1 Clamp Basics

In this section, we introduce the Clamp external language, in which dup and drop operations are implicit. The concrete syntax is borrowed from Haskell, but one prominent difference in Clamp is that each function type and term must be annotated with one of four substructural qualifiers: `U` for unlimited, `R` for relevant, `A` for affine, or `L` for linear.

Three examples of Clamp functions, translated from the Haskell standard prelude, appear in figure 1. Their types need not be written explicitly, and are inferred by Clamp's type checker.

Consider the `fst` function, which projects the first component of a pair. Because we would like be able to use library functions any number of times or not at all, we annotate the arrow in the lambda expression with qualifier `U`. This annotation determines the function type's structural properties—meaning, in this case, that `fst` satisfies both Dup and Drop. (Note that this is a property of the function *itself*, not of how it treats its argument.) Because `fst` does not use the second component of the pair, this induces the `Drop b` constraint on type variable `b`. In particular, elaboration into the internal language inserts a `drop` operation for `y` to make the term linear: `\(x, y) -U> drop y x`. The presence of `drop`, which disposes of its first argument and returns its second, causes the `Drop` type class constraint to be inferred.

Function `constU` imposes a similar constraint on its second argument, but it also requires the type of its first argument be unlimited. This is because `constU` returns an unlimited closure containing the first argument in its environment. The argument is effectively duplicated or discarded along with the closure, so it inherits the same structural restrictions. Alternatively, we can lift this restriction with `constL`, which returns a linear closure and thus allows the first argument to be linear.

$$
\begin{array}{rcll}
e & ::= & x \mid v \mid e_1\, e_2 \mid e\,[\overline{\tau_i}] \mid (e_1, e_2) \mid \textbf{letp}\ (x_1, x_2) = e\ \textbf{in}\ e_2 \mid \textbf{Inl}\ e \mid \textbf{Inr}\ e & \textit{(terms)} \\
 & & \mid \textbf{case}\ e\ \textbf{of Inl}\ x_1 \to e_1; \textbf{Inr}\ x_2 \to e_2 \mid \textbf{new}^{rq}\ e \mid \textbf{release}^{rq}\ e \mid \textbf{swap}^{rq}\ e_1\ \textbf{with}\ e_2 & \\
 & & \mid \textbf{dup}\ e_1\ \textbf{as}\ x_1, x_2\ \textbf{in}\ e_2 \mid \textbf{drop}\ e_1\ \textbf{in}\ e_2 & \\
v & ::= & \lambda^{aq} x{:}\tau.\, e \mid \Lambda\overline{\alpha_i}\,[P].\, v \mid (v_1, v_2) \mid \textbf{Inl}\ v \mid \textbf{Inr}\ v \mid \ell \mid () & \textit{(values)} \\
\tau & ::= & \alpha \mid \tau_1 \xrightarrow{aq} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathsf{Unit} \mid \mathsf{Ref}^{rq}\,\tau \mid \forall\overline{\alpha_i}.P \Rightarrow \tau & \textit{(types)} \\
P & ::= & (K_1\tau_1, \ldots, K_n\tau_n) & \textit{(constraints)} \\
rq & ::= & \mathsf{s}\ \textit{(strong)} \mid \mathsf{w}\ \textit{(weak)} & \textit{(reference qualifiers)} \\
aq & ::= & \mathsf{U}\ \textit{(unlimited)} \mid \mathsf{R}\ \textit{(relevant)} \mid \mathsf{A}\ \textit{(affine)} \mid \mathsf{L}\ \textit{(linear)} & \textit{(arrow qualifiers)} \\
K & ::= & \mathsf{Dup} \mid \mathsf{Drop} & \textit{(predicate constructors)}
\end{array}
$$

Figure 2: Syntax of $\lambda_{cl}$

# 2 Formalizing $\lambda_{cl}$

To validate the soundness of our approach, we have developed $\lambda_{cl}$, a core model of the Clamp internal language. $\lambda_{cl}$ is based on System F [11] with a few modifications: variable bindings are treated linearly, arrows are annotated with qualifiers, and type class constraints [12, 19] are added under universal quantifiers. As an example of how one can define custom usage-aware datatypes in Clamp, $\lambda_{cl}$ also includes a variety of operations for working with mutable references.

The $\lambda_{cl}$ type system shares many similarities with Tov's core Alms calculus: ${}^{a}\lambda_{ms}$ [17]. Unlike the external Clamp language prototype (§3), $\lambda_{cl}$ provides first-class polymorphism and does not support type inference.

## 2.1 Syntax of $\lambda_{cl}$

The syntax of $\lambda_{cl}$ appears in figure 2. Most of the language is standard, but notably arrow types and $\lambda$ terms in Clamp are annotated with an *arrow qualifier* (*aq*). These annotations determine which structural operations a function supports, as well as the corresponding constraints imposed on the types in its closure environment. Unlike some presentations of linear logic, $\xrightarrow{aq}$ here constrains usage of the function itself, not usage of the function's argument. Thus one can call dup on a $\xrightarrow{\mathsf{U}}$ arrow but not on an $\xrightarrow{\mathsf{L}}$ arrow. Type abstractions specify the type class constraints that they abstract over; their bodies are restricted to values, so unlike $\lambda$ terms, type abstractions do not need an arrow qualifier.

The $\textbf{new}^{rq}\ e$ and $\textbf{release}^{rq}\ e$ forms introduce and eliminate mutable references. Each comes in two flavors depending on its *reference qualifier* (*rq*), which records whether the reference supports strong or merely weak updates. Weak (conventional) updates must preserve a reference cell's type, but strong updates can modify both the value and type of a cell.

Form $\textbf{swap}^{rq}\ e_1\ \textbf{with}\ e_2$ provides linear access to a reference by exchanging its contents for a different value. The **release** operator deallocates a cell and returns its contents if it is not aliased. Store locations ($\ell$) appear at run time but are not written by the programmer.

To incorporate type classes, universal types may include constraints on their type variables. A constraint $P$ denotes a set of atomic predicate constraints $K\tau$, each of which is a predicate constructor $K$ applied to a type. For the sake of our current analysis, $K$ is either Dup or Drop.

$$
\begin{array}{lll}
E ::= [\cdot] \mid E\,e \mid v\,E \mid E\,[\overline{\tau_i}] \mid (E,e) \mid (v,E) \mid \mathbf{Inl}\,E \mid \mathbf{Inr}\,E & & \textit{(evaluation contexts)} \\
\quad \mid \mathbf{case}\,E\,\mathbf{of}\,\mathbf{Inl}\,x_1 \to e_1; \mathbf{Inr}\,x_2 \to e_2 \mid \mathbf{letp}\,(x_1,x_2) = E\,\mathbf{in}\,e \mid \mathbf{new}^{rq}\,E \mid \mathbf{release}^{rq}\,E & & \\
\quad \mid \mathbf{swap}^{rq}\,E\,\mathbf{with}\,e \mid \mathbf{swap}^{rq}\,v\,\mathbf{with}\,E \mid \mathbf{dup}\,E\,\mathbf{as}\,x_1,x_2\,\mathbf{in}\,e \mid \mathbf{drop}\,E\,\mathbf{in}\,e & & \\
\mu ::= \ell \mapsto^i v, \mu \mid \cdot & & \textit{(stores)}
\end{array}
$$

Figure 3: Runtime structures

$$
\begin{array}{rcll}
(\mu\ ;\ (\lambda x{:}\tau.\,e)\,v) & \longmapsto & (\mu\ ;\ \{v/x\}\,e) & \\
(\mu\ ;\ (\Lambda\overline{\alpha_i}\,[P]\,.\,v)\,[\overline{\tau_i}]) & \longmapsto & (\mu\ ;\ \overline{\{\tau_i/\alpha_i\}}v) & \\
(\mu\ ;\ \mathbf{new}^{rq}\,v) & \longmapsto & (\mu, \ell \mapsto^1 v\ ;\ \ell) & \ell \text{ fresh} \\
(\mu, \ell \mapsto^i v_1\ ;\ \mathbf{swap}^{rq}\,\ell\,\mathbf{with}\,v_2) & \longmapsto & (\mu, \ell \mapsto^i v_2\ ;\ (\ell, v_1)) & \\
(\mu, \ell \mapsto^1 v\ ;\ \mathbf{release}^{\mathbf{w}}\,\ell) & \longmapsto & (\mu\ ;\ \mathbf{Inl}\,v) & \\
(\mu, \ell \mapsto^i v\ ;\ \mathbf{release}^{\mathbf{w}}\,\ell) & \longmapsto & (\mu, \ell \mapsto^{i-1} v\ ;\ \mathbf{Inr}\,()) & \text{when } i > 1 \\
(\mu, \ell \mapsto^1 v\ ;\ \mathbf{release}^{\mathbf{s}}\,\ell) & \longmapsto & (\mu\ ;\ v) & \\
(\mu\ ;\ \mathbf{dup}\,v\,\mathbf{as}\,x_1,x_2\,\mathbf{in}\,e) & \longmapsto & (\mathrm{incr}(\mathrm{locs}\,(v)\,;\mu)\ ;\ \{v/x_2\}\{v/x_1\}\,e) & \\
(\mu\ ;\ \mathbf{drop}\,v\,\mathbf{in}\,e) & \longmapsto & (\mathrm{decr}(\mathrm{locs}\,(v)\,;\mu)\ ;\ e) & \\
(\mu_1\ ;\ E\,[e_1]) & \longmapsto & (\mu_2\ ;\ E\,[e_2]) & \text{when } (\mu_1\ ;\ e_1) \longmapsto (\mu_2\ ;\ e_2)
\end{array}
$$

Figure 4: Small-step relation

## 2.2 Semantics

The execution of $\lambda_{cl}$ terms can be defined by a call-by-value small-step semantics with evaluation contexts and a global, reference-counted store $\mu$. The run-time structures needed to define this small step relation are given in figure 3. Reference counts are used to track when reference cells can be safely deallocated in the presence of aliasing. Exchange properties for the store are implicitly assumed. A selection of the small step relation rules are given in figure 4, focusing on the rules for reference cells and substructural operations. Most of the complexity here comes from the reference counts.

The **swap** operator exchanges the contents of a cell in the heap with a different value. The **release** operator deallocates a cell and returns its contents if it is not aliased. However, if a cell has been aliased it decrements the reference count and returns a unit. The dup and drop operators manipulate reference counts as expected.

A few metafunctions given in figure 5 are necessary to maintain reference counts, similar to those in [7]. The functions $\mathrm{incr}(\ell;\mu)$ and $\mathrm{decr}(\ell;\mu)$ 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 reference cell involves decrementing the reference counts of all cells the deallocated contents originally pointed to.

The locs meta-function is a convenient way of extracting the multiset of locations that a value uses. Note that the $+$ (or $\uplus$) operator is a multiset operator which additively combines occurrences, and is used again in section 3.1. The locs function is also designed to operate on well-typed terms, so it only looks at one branch of a **case** expression, assuming that the other branch must share the same location typing context.

$$\mathrm{incr}(\ell;\ell\mapsto^{j} v,\mu) = \ell\mapsto^{j+1} v,\mu$$

$$\mathrm{decr}(\ell;\ell\mapsto^{j} v,\mu) = \begin{cases}\mathrm{decr}(\mathrm{locs}\,(v);\mu) & j=1\\ \ell\mapsto^{j-1} v,\mu & j>1\end{cases}$$

$$\mathrm{incr}(\{\ell_1,\ldots,\ell_k\};\mu) = \mathrm{incr}(\ell_1;\cdots\mathrm{incr}(\ell_k;\mu))$$

$$\mathrm{decr}(\{\ell_1,\ldots,\ell_k\};\mu) = \mathrm{decr}(\ell_1;\cdots\mathrm{decr}(\ell_k;\mu))$$

$$\mathrm{locs}\,(\ell) = \{\ell\}$$

$$\mathrm{locs}\,(\lambda^{aq}x{:}\tau.\,e) = \mathrm{locs}\,(e)$$

$$\mathrm{locs}\,(e_1\ e_2) = \mathrm{locs}\,(e_1)+\mathrm{locs}\,(e_2)$$

$$\mathrm{locs}\begin{pmatrix}\textbf{case}\ e\ \textbf{of}\\ \textbf{Inl}\ x_1\to e_1;\\ \textbf{Inr}\ x_2\to e_2\end{pmatrix} = \mathrm{locs}\,(e)+\mathrm{locs}\,(e_1)$$

$$\ldots$$

Figure 5: Reference count management

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

TABS
$$\frac{P_1,P_2;\Gamma;\Sigma\vdash v:\tau \qquad \mathrm{dom}\,P_2\subseteq\overline{\alpha_i}}{P_1;\Gamma;\Sigma\vdash\Lambda\overline{\alpha_i}\,[P_2].\,v:\forall\overline{\alpha_i}.P_2\Rightarrow\tau}$$

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

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

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

CASE
$$\frac{P;\Gamma_1;\Sigma_1\vdash e_1:\tau_{11}+\tau_{12} \qquad P;\Gamma_2,x_{21}{:}\tau_{11};\Sigma_2\vdash e_{21}:\tau_2 \qquad P;\Gamma_2,x_{22}{:}\tau_{12};\Sigma_2\vdash e_{22}:\tau_2}{P;\Gamma_1+\Gamma_2;\Sigma_1+\Sigma_2\vdash\textbf{case}\ e_1\ \textbf{of}\ \textbf{Inl}\ x_{21}\to e_{21};\ \textbf{Inr}\ x_{22}\to e_{22}:\tau_2}$$

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

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

Figure 6: Selected $\lambda_{cl}$ term typing rules

## 2.3 Term Typing

Variable contexts $\Gamma ::= x_1{:}\tau_1,\ldots,x_n{:}\tau_n$ associate variables with types, where each variable appears at most once. Location contexts (store typings) $\Sigma ::= \ell_s\mapsto_{\mathsf{s}}\tau_s,\ldots,\ell_w\mapsto^{k_w}_{\mathsf{w}}\tau_w,\ldots$ associate locations ($\ell$) with their reference types $\mathsf{Ref}\,\tau$, and distinguish between strong and weak locations; weak locations carry a reference count $k$ to track aliasing.

Linearity is enforced in $\lambda_{cl}$ via standard context-splitting. Because $\Gamma$ and $\Sigma$ are linear environments, we need operations to join them. The join operation $+$ is defined only on pairs of compatible environments, written $\Gamma_1\smile\Gamma_2$ and $\Sigma_1\smile\Sigma_2$. Two variable contexts are compatible so long as they are disjoint. Two location contexts are compatible if the strong locations are disjoint and the weak locations in their intersection agree on their types. Joining variable contexts appends the two sets of bindings together, while joining location contexts also involves adding the reference counts of any shared weak locations. Contexts are identified up to permutation.

The term typing judgment ($P;\Gamma;\Sigma\vdash e:\tau$) assigns term $e$ type $\tau$ under constraint, variable, and location contexts $P$, $\Gamma$, and $\Sigma$. Selected typing rules for the core language appear in figure 6, and the

$$\textsc{New}\ \frac{P;\Gamma;\Sigma \vdash e : \tau}{P;\Gamma;\Sigma \vdash \mathbf{new}^{rq}\ e : \mathsf{Ref}^{rq}\ \tau} \qquad \textsc{LocW}\ \frac{}{P;\cdot;\ell \mapsto^{1}_{\mathsf{w}} \tau \vdash \ell : \mathsf{Ref}^{\mathsf{w}}\ \tau} \qquad \textsc{LocS}\ \frac{}{P;\cdot;\ell \mapsto_{\mathsf{s}} \tau \vdash \ell : \mathsf{Ref}^{\mathsf{s}}\ \tau}$$

$$\textsc{ReleaseW}\ \frac{P;\Gamma;\Sigma \vdash e : \mathsf{Ref}^{rq}\ \tau}{P;\Gamma;\Sigma \vdash \mathbf{release}^{w}\ e : \mathsf{Unit} + \tau} \qquad \textsc{ReleaseS}\ \frac{P;\Gamma;\Sigma \vdash e : \mathsf{Ref}^{\mathsf{s}}\ \tau}{P;\Gamma;\Sigma \vdash \mathbf{release}^{s}\ e : \tau}$$

$$\textsc{SwapW}\ \frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \mathsf{Ref}^{rq}\ \tau \qquad P;\Gamma_2;\Sigma_2 \vdash e_2 : \tau}{P;\Gamma_1+\Gamma_2;\Sigma_1+\Sigma_2 \vdash \mathbf{swap}^{\mathsf{w}}\ e_1\ \mathbf{with}\ e_2 : \mathsf{Ref}^{rq}\ \tau \times \tau}$$

$$\textsc{SwapS}\ \frac{P;\Gamma_1;\Sigma_1 \vdash e_1 : \mathsf{Ref}^{\mathsf{s}}\ \tau_1 \qquad P;\Gamma_2;\Sigma_2 \vdash e_2 : \tau_2}{P;\Gamma_1+\Gamma_2;\Sigma_1+\Sigma_2 \vdash \mathbf{swap}^{\mathsf{s}}\ e_1\ \mathbf{with}\ e_2 : \mathsf{Ref}^{\mathsf{s}}\ \tau_2 \times \tau_1}$$

Figure 7: $\lambda_{cl}$ term reference cell typing rules

typing rules for reference cells are given in figure 7. Consistency conditions $\Sigma_1 \smile \Sigma_2$ and $\Gamma_1 \smile \Gamma_2$ are assumed whenever contexts are combined. The core language typing rules split and share the linear contexts as needed, but are otherwise a natural extension of System F to support type class constraints.

We impose a syntactic restriction, similar to Haskell 98's context reduction restrictions [15], on the form of constraints in type schemes introduced by the TABS rule: type abstractions may only constrain the type variables that they bind, and not compound or unrelated types. This simplifies induction over typing derivations for TABS since it means that no constraints on external type variables can be introduced by a type abstraction. Additionally, in rule ABS, the variable and location contexts are constrained by the function's arrow qualifier, to ensure that values captured by the closure support any structural operations that might be applied to the closure itself; this constraint must be entailed ($\Vdash$) by the constraint context. Here $\mathsf{Constrain}^{aq}(\Gamma;\Sigma)$ is shorthand for the appropriate set of Dup and Drop constraints applied to every type mapped in $\Gamma$ and $\Sigma$, so that for instance $\mathsf{Constrain}^{\mathsf{L}}$ imposes no constraints, while $\mathsf{Constrain}^{\mathsf{R}}$ imposes only Dup constraints.

The **dup** and **drop** forms constrain the types of their parameters in the expected way, by requiring their types to be members of the Dup or Drop type classes, respectively (again entailed by the constraint context).

Since $\lambda_{cl}$ supports both strong and weak references with different substructural properties, there are a variety of typing rules governing their usage. The **swap** operation needs to return both an updated reference and the old contents, so it packages those in a pair. Weak and strong forms of reference cell operations are provided, and it is safe to apply the weak operations to both strong and weak references. The $\mathbf{release}^{rq}\ e$ 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, we preserve its linearity while allowing for aliasing by returning the contents of the reference only when the last alias to the cell is released, and unit otherwise.

$$(\mathsf{Dup}\,\alpha_1, \mathsf{Dup}\,\alpha_2) \Rightarrow \mathsf{Dup}\,(\alpha_1 \times \alpha_2) \qquad (\mathsf{Drop}\,\alpha_1, \mathsf{Drop}\,\alpha_2) \Rightarrow \mathsf{Drop}\,(\alpha_1 \times \alpha_2)$$
$$(\mathsf{Dup}\,\alpha_1, \mathsf{Dup}\,\alpha_2) \Rightarrow \mathsf{Dup}\,(\alpha_1 + \alpha_2) \qquad (\mathsf{Drop}\,\alpha_1, \mathsf{Drop}\,\alpha_2) \Rightarrow \mathsf{Drop}\,(\alpha_1 + \alpha_2)$$

$$() \Rightarrow \mathsf{Dup}\,(\alpha_1 \xrightarrow{\mathsf{U}} \alpha_2) \quad () \Rightarrow \mathsf{Drop}\,(\alpha_1 \xrightarrow{\mathsf{U}} \alpha_2) \quad () \Rightarrow \mathsf{Dup}\,\mathsf{Unit} \qquad () \Rightarrow \mathsf{Drop}\,\mathsf{Unit}$$
$$() \Rightarrow \mathsf{Dup}\,(\alpha_1 \xrightarrow{\mathsf{R}} \alpha_2) \quad () \Rightarrow \mathsf{Drop}\,(\alpha_1 \xrightarrow{\mathsf{A}} \alpha_2) \quad () \Rightarrow \mathsf{Dup}\,(\mathsf{Ref}^{\mathsf{w}}\,\alpha) \quad (\mathsf{Drop}\,\alpha) \Rightarrow \mathsf{Drop}(\mathsf{Ref}^{rq}\,\alpha)$$

Figure 8: Dup and Drop instances

## 2.4 Type Class Instances

Throughout the type system, type class constraints are propagated via entailment, $P_1 \Vdash P_2$, which specifies when one set of type class predicates ($P_2$) is implied by another ($P_1$) in the context of the fixed background instance environment $\Gamma^{\text{is}}$. For example, entailment allows our type system to derive that $\mathsf{Unit} \times \mathsf{Unit}$ is duplicable because Unit is. Rules for entailment are given by Jones [12] and adapt naturally to this setting. The substructural essence of the type class system in Clamp is the set of base Dup and Drop instances $\Gamma^{\text{is}}$, which appears in figure 8.

Since pairs and sums contain values that might be copied or ignored along with the pair or sum value, their instance rules require instances for their components. Functions impose constraints on their closure environments when they are assigned a qualifier during term typing, so the instance rules for arrows depend only on the arrow qualifier.

Dealing correctly with references is more subtle, as seen in $\lambda^{\mathsf{refURAL}}$ [2]. In Clamp, some references support strong updates, which can change not only the value but the type of a mutable reference. However it is unsafe to alias a reference cell whose type might change.

In $\lambda^{\mathsf{refURAL}}$, the restrictions on reference types are given in a sizable table, but Dup and Drop instances make it easy to express these restrictions in Clamp. Clamp classifies references by the kind of updates they support: strong or weak. This is specified by the *rq* qualifier in the $\mathsf{Ref}^{rq}$ type.

Qualitatively, the constraints we impose are that:

- Strong references may not be duplicated.
- Only references with droppable contents may be dropped.
- Only strong references support direct deallocation.
- Weak references can be deallocated, but only return their contents when unaliased.

The above four rules capture the same restrictions as $\lambda^{\mathsf{refURAL}}$ references. They also increase the expressiveness of the system by explicitly distinguishing weak and strong references and allowing for the deallocation of weak references. They are expressed in $\lambda_{cl}$ with two type class instances and the typing judgments RELEASEW and RELEASES.

As an example of the kinds of structures we can build using these rules, consider the type

$$\mathsf{Ref}^{\mathsf{w}}\,(\text{fhandle})$$

for a linear file handle fhandle. This weak reference can be aliased to provide shared access to the file handle, but cannot be dropped based on the type class instances, since fhandle cannot be dropped. Anyone that uses this reference must release the reference and close the file if necessary.

## 2.5 Type Safety

Here we sketch part of the type safety proof; more details may be found in Gan's thesis [9].

The bulk of the work goes into proving preservation, and the key lemma in proving preservation relates constraints to bindings. Intuitively, this lemma says that structural constraints on a value's type respect the structural constraints of everything the value contains or points to, via the variable and location contexts. Syntactic forms like $\mathsf{Dup}\,\Gamma$ are used to denote the set of Dup constraints on all types in $\Gamma$, and similarly $\mathsf{Dup}\,\Sigma$ applies Dup to all of the $\mathsf{Ref}^{\mathrm{rq}}\,\tau$ types mapped by $\Sigma$. Note that the lemma does not hold for arbitrary expressions.

**Lemma 1** (Constraints capture bindings)**.** *Suppose that* $P;\Gamma;\Sigma \vdash v : \tau$. *If* $P \Vdash \mathsf{Dup}\,\tau$ *then* $P \Vdash (\mathsf{Dup}\,\Sigma, \mathsf{Dup}\,\Gamma)$*; if* $P \Vdash \mathsf{Drop}\,\tau$ *then* $P \Vdash (\mathsf{Drop}\,\Sigma, \mathsf{Drop}\,\Gamma)$.

*Proof.* By induction on the typing derivation for $v$. □

Lemma 1 is essential to proving the substitution lemma (Lemma 2).

**Lemma 2** (Substitution)**.** *If*

- $P;\Gamma,x{:}\tau_x;\Sigma_1 \vdash e : \tau$,
- $P;\cdot;\Sigma_2 \vdash v : \tau_x$, *and*
- $\Sigma_1 \smile \Sigma_2$,

*then* $P;\Gamma;\Sigma_1 + \Sigma_2 \vdash \{v/x\}\,e : \tau$

*Proof.* By induction on the typing derivation for $e$, making use of lemma 1 in the $\lambda$ case. □

In proving Preservation, it is also useful to separate out a Replacement Lemma which specifies exactly how substitution interacts with evaluation contexts.

**Lemma 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* $\Gamma = \Gamma_1 + \Gamma_2$ *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* $P;\Gamma_1' + \Gamma_2;\Sigma_1' + \Sigma_2 \vdash E\,[M'] : \tau$ *for any* $M', \Gamma_1', \Sigma_1'$

*Proof.* By induction on $E$. □

Another key lemma for proving preservation relates the locs function used to maintain dynamic reference counts with the store context that a value requires. To state this lemma, we overload the locs function to also return the multiset of occurrences (multiple for reference counted weak location stores) of locations in the domain of a store context.

**Lemma 4** (Store Contexts map Free Locations)**.** *If* $P;\Gamma;\Sigma \vdash e : \tau$ *then* $\mathrm{locs}\,(e) = \mathrm{locs}\,(\Sigma)$.

Finally, to prove preservation and type soundness we need to introduce store and configuration typings which are given in Figure 9. The remainder of the type soundness proof is then mostly standard.

**Lemma 5** (Preservation)**.** *If* $\vdash_c (\mu_1\ ;\ e_1) : \tau$ *and* $(\mu_1\ ;\ e_1) \longmapsto (\mu_2\ ;\ e_2)$ *then* $\vdash_c (\mu_2\ ;\ e_2) : \tau$

**Theorem 1** (Type soundness)**.** *If* $\vdash_c (\cdot\ ;\ e) : \tau$ *then either it diverges or it reduces to a value configuration* $(\mu\ ;\ v)$ *such that* $\vdash_c (\mu\ ;\ v) : \tau$.

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

$$\textsc{Conf}\ \frac{\Sigma_1 \vdash_s \mu : \Sigma_1 + \Sigma_2 \qquad \cdot;\cdot;\Sigma_2 \vdash e : \tau}{\vdash_c (\mu\ ;\ e) : \tau}$$

Figure 9: Store and configuration typing

# 3 Implementing the Clamp Type Checker

We have implemented a type checker that infers Damas–Hindley–Milner style type schemes for Clamp terms. The type checker is an extension of Jones's "Typing Haskell in Haskell" type checker [13]. Its source code may be found at `https://github.com/edgan8/clampcheck`.

The process of modifying a Haskell type checker to support Clamp was straightforward and illustrates one of the strengths of Clamp's design: It requires only small and orthogonal additions to a language like Haskell. Besides adding qualifiers to arrow types, we made three additions to a Haskell type checker:

1. an elaboration pass that inserts dups and drops,
2. Dup and Drop type classes and instances, and
3. substructural qualifiers and constraints on arrow types.

## 3.1 Inferring dups and drops.

The elaboration pass is the bridge between a concise user-facing language and leveraging conventional, nonlinear type checking techniques. The pass takes as input a term with arbitrary variable usages; it inserts the appropriate dup and drop operations and renames the duplicated copies so that in the resulting term all variable usage is strictly linear. Structural properties are then enforced by the constraints imposed by dup and drop.

Since different elaborations can lead to different static and dynamic semantics, we have proven that our algorithm generates an *optimal* elaboration in two senses:

- It minimizes the program's live variables.
- It imposes minimal type class constraints.

In what follows we define a core linear language to formalize and estabilsh these two points.

**An Abstract Linear Language** To focus on the essential problems, we can work with an *abstraction* of the linear $\lambda$ calculus, $\hat{\lambda}_{lin}$. By modeling only usage and binding, $\hat{\lambda}_{lin}$ allows us to focus on inserting dup and drop operations independently of the particular types and term forms of a language. Its syntax is given in figure 10. Extending the results to cover other term forms is straightforward.

The product expression $e_1 \otimes e_2$ abstracts multiplicative forms such as pairs and function applications—that is, pairs of expressions where both will be evaluated. The sum expression $e_1 \,\&\, e_2$ abstracts additive forms such as linear logic's additive conjuction, and the relationship between the branches of a **case**—that is, pairs of expressions where exactly one will be evaluated.

$$e ::= x \mid \lambda x.e \mid e_1 \otimes e_2 \mid e_1 \,\&\, e_2 \qquad \textit{(unannotated terms)}$$
$$ae ::= x \mid \lambda x.ae \mid ae_1 \otimes ae_2 \mid ae_1 \,\&\, ae_2 \mid \textbf{dup}\ \Gamma\ \textbf{in}\ ae \mid \textbf{drop}\ \Gamma\ \textbf{in}\ ae \qquad \textit{(annotated terms)}$$

Figure 10: $\hat{\lambda}_{lin}$ syntax

$$\text{L-VAR}\ \frac{}{\{x\} \vdash x} \qquad \text{L-ABS}\ \frac{\Gamma + \{x\} \vdash ae \qquad x \notin \Gamma}{\Gamma \vdash \lambda x.ae} \qquad \text{L-PAIR}\ \frac{\Gamma_1 \vdash ae_1 \qquad \Gamma_2 \vdash ae_2}{\Gamma_1 + \Gamma_2 \vdash ae_1 \otimes ae_2} \qquad \text{L-CHOICE}\ \frac{\Gamma \vdash ae_1 \qquad \Gamma \vdash ae_2}{\Gamma \vdash ae_1 \,\&\, ae_2}$$

$$\text{L-DUP}\ \frac{\Gamma_1 + \Gamma_2 + \Gamma_2 \vdash ae}{\Gamma_1 + \Gamma_2 \vdash \textbf{dup}\ \Gamma_2\ \textbf{in}\ ae} \qquad \text{L-DROP}\ \frac{\Gamma_1 \vdash ae}{\Gamma_1 + \Gamma_2 \vdash \textbf{drop}\ \Gamma_2\ \textbf{in}\ ae}$$

Figure 11: Annotated expression well-formedness

Expressions, $e$, are unannotated and don't explicitly satisfy linear usage constraints. Annotated expression, $ae$, use dup and drop operations to explicitly specify nonlinear usage of variables. The dup and drop operations work over contexts, $\Gamma$, which are multisets of variables

The contexts $\Gamma$ in $\hat{\lambda}_{lin}$ manage scope and binding by restricting contraction and weakening to explicit dup and drop annotations, but do not track the types of variables. In order to avoid the messy but straightforward process of generating names and renaming variables when inserting a dup, we think of contexts as multisets (*e.g.,* $\{x,x,y,z,\ldots\}$) of in-scope variables, or equivalently as functions from variables to natural numbers. Thus $\Gamma(x)$ will be used to denote the number of times $x$ appears in $\Gamma$.

We use these multiset contexts to define a notion of *well-formedness* in figure 11, which describes when an annotated term $ae$ in $\hat{\lambda}_{lin}$ properly accounts for all nonlinear usage of its variables through explicit dup and drop operations.

**Inference Algorithm** An inference algorithm for annotating terms is given in figure 12. The strategy is to recursively transform a term bottom-up based on the free variables fv in each recursively transformed sub-term. Note that the fv function always returns a *set* and that $\cap$ and $\setminus$ denote the standard set intersection and difference operators.

Dup operations are inserted where the free variables of two sub-terms of a multiplicative form (*e.g.,* application, but not branching) are discovered to intersect; drops are added under binders when the bound variable is not free in its scope, and when a variable used in one branch (say, of an if-then-else) is not free in the other.

We outline the key steps in our optimality argument here; additional details may be found in [9]. Lemma 6 states that the algorithm is sound.

**Lemma 6** (Soundness)**.** $\text{fv}(e) \vdash \text{infer}(e)$

If $ae$ is an annotation of $e$, then $\text{fv}(e) \sqsubseteq \text{fv}(ae)$, so lemma 7 shows that our algorithm in fact generates an annotation which requires a minimal context $\Gamma$ for well-formedness.

**Lemma 7** (Minimal contexts)**.** *If* $\Gamma \vdash ae$ *then* $\text{fv}(ae) \sqsubseteq \Gamma$.

With some technical lemmas, we can then prove that the algorithm introduces no unnecessary dups or drops on variables. In order to compare different potential annotations of the same term, we de-

$$\begin{aligned}
\text{infer}(x) &= x \\
\text{infer}(\lambda x.e) &= \begin{cases} \lambda x.\,\text{infer}(e) & \text{if } x \in \text{fv}(e); \\ \lambda x.\mathbf{drop}\ x\ \mathbf{in}\ \text{infer}(e) & \text{otherwise} \end{cases} \\
\text{infer}(e_1 \otimes e_2) &= \mathbf{dup}\ \text{fv}(e_1) \cap \text{fv}(e_2)\ \mathbf{in}\ \text{infer}(e_1) \otimes \text{infer}(e_2) \\
\text{infer}(e_1 \,\&\, e_2) &= (\mathbf{drop}\ \text{fv}(e_2) \setminus \text{fv}(e_1)\ \mathbf{in}\ \text{infer}(e_1)) \,\&\, (\mathbf{drop}\ \text{fv}(e_1) \setminus \text{fv}(e_2)\ \mathbf{in}\ \text{infer}(e_2))
\end{aligned}$$

Figure 12: Inference algorithm

fine a function erase : $ae \to e$, which removes dup and drop annotations from an annotated term in the straightforward way, yielding an unannotated term. It should be evident that $\text{erase}(\text{infer}(e)) = e$.

**Lemma 8** (Forced drop)**.** *If* $\Gamma \vdash ae$, $\Gamma(x) \geq 1$, *and* $x \notin \text{fv}(\text{erase}(ae))$, *then ae contains a subterm* **drop** $\Gamma'$ **in** $ae'$ *such that* $x \in \Gamma'$.

**Lemma 9** (Forced dup)**.** *If* $\Gamma \vdash ae$, $\Gamma(x) \leq 1$, *and there exists a subderivation* $\Gamma_s \vdash ae_s$ *of* $\Gamma \vdash ae$ *with* $\Gamma_s(x) \geq 2$, *then ae contains a subterm* **dup** $\Gamma'$ **in** $ae'$ *such that* $x \in \Gamma'$

**Lemma 10** (No Unnecessary Drops)**.** *Let* $ae = \text{infer}(e)$. *If ae contains a subterm* **drop** $\Gamma_d$ **in** $ae_s$ *with* $x \in \Gamma_d$ *then any other well-formed* $ae'$ *with* $\text{erase}(ae') = e$ *contains a subterm* **drop** $\Gamma_d'$ **in** $ae_s'$ *with* $x \in \Gamma_d'$.

**Lemma 11** (No Unnecessary Dups)**.** *Let* $ae = \text{infer}(e)$. *If ae contains a subterm* **dup** $\Gamma_d$ **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* **dup** $\Gamma_d'$ **in** $ae_s'$ *with* $x \in \Gamma_d'$.

### 3.2 Constraint processing.

After the dup and drop insertion pass, type inference can proceed without needing to count variable usages or split contexts, since the insertion pass has made every dup and drop operation explicit. With the exception of the extra constraints imposed on closure environments, inferring types for the Clamp internal language is like inferring types for Haskell. For the constraint solver, the type classes Dup and Drop and their instances are no different than any other type class.

Type checking in this system is thus separated into two self-contained steps: first, usage analysis as performed by elaboration, and second, checking substructural constraints in the same manner as any other type class system. A similar division was used by de Vries et al. to integrate a uniqueness typing system into Damas–Hindley–Milner [18].

In Table 1 we present the sizes of the components of our Clamp implementation. It compares favorably to the implementation of languages such as Alms [17], whose type inference engine is 15,000 lines of Haskell. The dup/drop insertion sits on top of the stack and is the main addition we have had to make to a Haskell type checker design. Besides that, we have included the base set of type class instances and altered arrow kinds throughout.

## 4 Related Work

Of the existing work in linear type systems, we will focus here on those which develop general purpose polymorphic linear types. Research on the mathematical expressiveness of linearity [3] or more tailored

| Component | Lines of code |
|---|---|
| Dup/drop insertion | 160 |
| Type class instances | 60 |
| Syntax and Types | 703 |
| Parser/Lexer | 319 |
| Unification Engine | 373 |

Table 1: Type checker code breakdown

use cases [10] for instance often do not aim at broad usability and polymorphism.

The first linear type systems derive directly from intuitionistic linear logic, and use the exponential "!" to indicate types that support structural operations [1, 6]. Some later type systems, in order to support parametric polymorphism over linearity, replace "!" with types composed of a qualifier and a pretype [2, 20], so that all types in these languages have a form like $^{q}\overline{\tau}$. Similarly, the Clean programming language makes use of qualifier variables and inequalities to capture a range of substructural polymorphism [5, 18]. Though Clean uses uniqueness rather than linear types, many of its design decisions can be applied in a linear settings as well.

More recent languages such as Alms [17] and $F^{\circ}$ [14] eliminate the notational overhead of annotating every type with substructural qualifiers by using distinct kinds to separate substructural types. Thus, rather than working with types like $^{\mathsf{A}}\mathsf{file}$, a file type in Alms can be defined to have kind A. Like Clean, Alms is highly polymorphic, but it makes use of compound qualifier expressions on function types, as well as dependent kinds and sub-kinding.

Compared to type systems like those of Clean and Alms, we believe Clamp offers advantages in simplicity and extensibility. Like Alms and $F^{\circ}$, Clamp avoids the burden in Clean of annotating every type with a qualifier. Type classes themselves are a general and powerful feature; for a language that is going to have type classes anyway, the Clamp approach allows adding the full spectrum of URAL types with little additional complexity for programmers and type checkers. Programmers already familiar with type classes will be well prepared to understand Clamp-style substructural types.

Further, type classes provide a clean formalism for constraining state-aware datatypes such as the system of weak and strong mutable references found in Clamp (§2.4). Finally, we anticipate that user-defined `Dup` and `Drop` instances, not yet supported by Clamp, will allow defining custom destructors and copy constructors, which should enable a variety of resource management strategies.

However, compared to Alms and Clean, Clamp does not provide as much polymorphism because each arrow is assigned a concrete qualifier. Consider, for instance, a *curry* function in Clamp. Unlike in Alms or Clean, Clamp requires different versions for different desired structural properties. For instance, two possible type schemes for a *curry* function are

```
(Dup a, Drop a) => ((a, b) -U> c) -U> a -U> b -U> c
```

and

```
((a, b) -L> c) -U> a -L> b -L> c.
```

We believe that extending Clamp with qualifier variables and type class implications could increase its expressiveness to the point where *curry* has a principal typing.

# 5 Future Work

## 5.1 Custom `dup` and `drop`

In Clamp's current design, the semantics of dup and drop are fixed. Allowing programmers instead to define their own implementations of dup and drop on user-defined types would enable scenarios similar to those possible in C++ via copy constructors and destructors. Programmers could then define data types that automatically manage resources in ways that meet particular needs; but unlike in C++, we believe this could be done without unsafe operations.

Programmers could define types and instances to manage their memory in whatever way is most appropriate, for instance by choosing between deep-copy and shallow-copy dup operations (or perhaps some hybrid approach), or between eager and lazy drop operations. For such a system to be practical, it is important that the dup/drop-insertion algorithm be easy to understand, since the insertion of dups and drops can affect the dynamic semantics of the language.

## 5.2 Polymorphic Arrows

In most cases, Clamp allows programmers to define functions that are inherently polymorphic over the substructural properties of their arguments. In these cases, a function that 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 function types, which are annotated with fixed qualifiers determined at the function definition point.

Alms is able to accommodate more polymorphic arrow types by introducing a subtyping relation on qualified arrows [17], and it refines the types of arrows further by introducing usage qualifiers that depend on the substructural properties of type variables. This allows one to write a function whose qualifier is inferred from the closure environment and inherits any polymorphism present in that environment.

There are many options for increasing the polymorphic expressiveness of Clamp without resorting to the complexities of subtyping. One could make qualifiers into first class types and allow quantification over qualifiers in arrow types. This is implemented in an ad-hoc way in our current type checker.

The idea of expanding the language of qualifiers could also be profitable in Clamp. This often means annotating arrow types with the types of their closure environments, and in a sense assigning them a closure-converted type. The dup and drop instances for arrows could then use the closure environment types to determine the arrow type's substructural properties.

## 5.3 Implementation

The current implementation of the Clamp type checker reflects $\lambda_{cl}$, and could benefit from the addition of some standard language features found in full Haskell. In particular, the addition of algebraic datatypes, user-defined instance rules, and a module system would allow programmers to define libraries that expose custom types with varying substructural properties.

It would also be interesting to implement a compiler for Clamp that takes advantage of substructural properties for reference-counted memory management [7]. Eagerly reusing the storage occupied by linear and affine values may offer particular performance advantages, and substructural analyses can enable a variety of other optimizations as well [4, 21].

## 6 Conclusion

Clamp introduces techniques that make it easier and more desirable to add substructural types to functional programming languages. The external / internal language distinction gives us both a programmer friendly syntax and a direct path to type inference. The Dup and Drop classes also support polymorphism over the URAL lattice and can represent state aware types such as strong and weak references. Type classes are an expressive and well-established language feature, and Clamp shows that they can serve as a base for substructural types.

## References


[1] Samson Abramsky (1993): *Computational Interpretations of Linear Logic*. *Theor. Comput. Sci.* 111(1-2), doi:10.1016/0304-3975(93)90181-R.

[2] Amal Ahmed, Matthew Fluet & Greg Morrisett (2005): *A Step-Indexed Model of Substructural State*. In: *Proc. 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, doi:10.1145/1086365.1086376.

[3] Sandra Alves, Maribel Fernández, Mário Florido & Ian Mackie (2011): *Linearity and Recursion in a Typed Lambda-calculus*. In: *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, ACM, New York, NY, USA, pp. 173–182, doi:10.1145/2003476.2003500.

[4] Henry G. Baker (1994): *A "linear logic" Quicksort*. *SIGPLAN Not.* 29(2), pp. 13–18, doi:10.1145/181748.181750.

[5] Erik Barendsen & Sjaak Smetsers (1996): *Uniqueness Typing for Functional Languages with Graph Rewriting Semantics*. In: *Math. Struct. Comp. Sci.*

[6] Gavin M. Bierman (1993): *On Intuitionistic Linear Logic*. Ph.D. thesis, University of Cambridge.

[7] Jawahar Chirimar, Carl A. Gunter & Jon G. Riecke (1996): *Reference Counting as a Computational Interpretation of Linear Logic*. *Journal of Functional Programming* 6, pp. 6–2, doi:10.1017/S0956796800001660.

[8] Luis Damas & Robin Milner (1982): *Principal Type-Schemes for Functional Programs*. In: *Proc. 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82)*, doi:10.1145/582153.582176.

[9] Edward Gan (2013): *Clamp: Type Classes for Substructural Types*. Senior Thesis, Harvard University. Available at `http://edgan8.github.io/`.

[10] Simon J. Gay & Vasco T. Vasconcelos (2010): *Linear type theory for asynchronous session types*. *Journal of Functional Programming* 20, pp. 19–50, doi:10.1017/S0956796809990268.

[11] Jean-Yves Girard (1972): *Interprétation fonctionnelle et Elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7, doi:10.1016/0304-3975(87)90045-4.

[12] Mark P. Jones (1995): *Qualified Types: Theory and Practice*. Cambridge University Press, New York.

[13] Mark P. Jones (1999): *Typing Haskell in Haskell*. In: *Proc. 1999 Haskell Workshop*.

[14] Karl Mazurak, Jianzhou Zhao & Steve Zdancewic (2010): *Lightweight Linear Types in System $F^\circ$*. In: *Proc. 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, doi:10.1145/1708016.1708027.

[15] Simon Peyton Jones & John Hughes, ed. (1999): *Haskell 98: A Non-Strict, Purely Functional Language*.

[16] Jesse A. Tov (2012): *Practical Programming with Substructural Types*. Ph.D. thesis, Northeastern University.

[17] Jesse A. Tov & Riccardo Pucella (2011): *Practical Affine Types*. In: *Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, doi:10.1145/1926385.1926436.

[18] Edsko Vries, Rinus Plasmeijer & David M. Abrahamson (2008): *Implementation and Application of Functional Languages*. chapter Uniqueness Typing Simplified, Springer-Verlag, Berlin, Heidelberg, pp. 201–218, doi:10.1007/978-3-540-85373-2_12.

[19] Philip Wadler & Stephen Blott (1989): *How to Make Ad-Hoc Polymorphism Less Ad Hoc*. In: *Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*, doi:10.1145/75277.75283.

[20] David Walker (2005): *Advanced Topics in Types and Programming Languages*, chapter 1. MIT Press, Cambridge, Mass., U.S.A.

[21] Keith Wansbrough & Simon Peyton Jones (1999): *Once upon a Polymorphic Type*. In: *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, ACM, pp. 15–28, doi:10.1145/292540.292545.