Type Classes for Lightweight Substructural Types

Edward Gan[1], Jesse A. Tov[2], and Greg Morrisett[2]

[1] Facebook, Menlo Park, Calif., U.S.A.
edgan@fb.com
[2] Harvard University, Cambridge, Mass., U.S.A.
{tov, greg}@eecs.harvard.edu

Abstract

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 for Clamp in Haskell.

1 Introduction

Type classes [8, 15] provide a way to constrain types by which operations they support. If the type class predicate $\mathsf{Dup}\,\alpha$ indicates when assumptions of type α 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 [12], forms of the basis of the type system for our prototype substructural programming language Clamp.

The semantics of Clamp are given in terms of a syntactically linear internal language in which all variables are used exactly once. To copy and discard values in the internal language, Clamp provides *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:

$$\begin{aligned} dup &: \forall\alpha.\, \mathsf{Dup}\,\alpha \Rightarrow \alpha \to \alpha \otimes \alpha \\ drop &: \forall\alpha\beta.\, \mathsf{Drop}\,\alpha \Rightarrow \alpha \to \beta \to \beta \end{aligned}$$

Clamp programs are written in an ML-like external language in which weakening and contraction are implicit, and then elaborated into the internal language through insertion of dups and drops.

Thus, in the internal language all nonlinear usage must be mediated through 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

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

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 [5] with type classes [8, 15]; improper duplication and dropping is indicated by unsatisfiable type class constraints.

```
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. The specific contributions in this paper include:

- a type system design with polymorphic substructural types and a type safety theorem (§2);
- a simple system for managing weak and strong references (§2.3);
- a type checker for Clamp derived directly from a type checker for Haskell (§3); and
- a dup-and-drop–insertion algorithm that is in some sense optimal (§3).

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.

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

Figure 2: Syntax of λ_{cl}

2 Formalizing the Type System

To validate the soundness of our approach, we have developed λ_{cl}, a core model of the Clamp internal language. λ_{cl} is based on System F [7] with a few modifications: variable bindings are treated linearly, arrows are annotated with qualifiers, and type class constraints [8, 15] are added under universal quantifiers. The λ_{cl} type system also shares many similarities with Tov's ${}^{a}\lambda_{ms}$ [13]. Unlike the external Clamp language prototype (§3), λ_{cl} provides first-class polymorphism and does not support type inference.

2.1 Syntax of λ_{cl}

The syntax of λ_{cl} appears in figure 2. Most of the language is standard, but notably arrow types and λ terms in Clamp are annotated with an *arrow qualifier* (aq). These annotations determine which structural operations a function allows, 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. Type abstractions specify the type class constraints that they abstract over; their bodies are restricted to values, so unlike λ terms, type abstractions do not need an arrow qualifier.

The **new**^{rq} e and **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. Form **swap**^{rq} e_1 **with** e_2 provides linear access to a reference by exchanging its contents for a different value. Store locations (ℓ) 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.

2.2 Term Typing

Variable contexts Γ associate variables with types, where each variable appears at most once. Location contexts (store typings) Σ associate locations (ℓ) with their types, and distinguish between strong and weak locations (not shown); weak locations carry a reference count to track aliasing.

$$\textsc{Var}\quad \frac{}{P; x{:}\tau; \cdot \vdash x : \tau} \qquad \textsc{TAbs}\quad \frac{P_1, P_2; \Gamma; \Sigma \vdash v : \tau \qquad \mathrm{dom}\, P_2 \subseteq \overline{\alpha_i}}{P_1; \Gamma; \Sigma \vdash \Lambda\overline{\alpha_i}\,[P_2]\,.\,v : \forall\overline{\alpha_i}.P_2 \Rightarrow \tau}$$

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

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

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

$$\textsc{Dup}\quad \frac{\begin{array}{c} P; \Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \\ P; \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_1; \Sigma_2 \vdash e_2 : \tau_2 \qquad P \Vdash \mathsf{Dup}\,\tau_1 \end{array}}{P; \Gamma_1 + \Gamma_2; \Sigma_1 + \Sigma_2 \vdash \mathbf{dup}\ e_1\ \mathbf{as}\ x_1, x_2\ \mathbf{in}\ e_2 : \tau_2} \qquad \textsc{Drop}\quad \frac{\begin{array}{c} P; \Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \\ P; \Gamma_2; \Sigma_2 \vdash e_2 : \tau_2 \qquad P \Vdash \mathsf{Drop}\,\tau_1 \end{array}}{P; \Gamma_1 + \Gamma_2; \Sigma_1 + \Sigma_2 \vdash \mathbf{drop}\ e_1\ \mathbf{in}\ e_2 : \tau_2}$$

Figure 3: Selected λ_{cl} term typing rules

Linearity is enforced in λ_{cl} via standard context-splitting. Because Γ and Σ 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 τ under constraint, variable, and location contexts P, Γ, and Σ. Selected type inference rules appear in figure 3. 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 [11], 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. 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 $\mathrm{Constrain}^{aq}(\Gamma;\Sigma)$ is shorthand for the appropriate set of Dup and Drop constraints applied to every type appearing in Γ and Σ, so that for instance $\mathrm{Constrain}^{\mathsf{L}}$ imposes no constraints, while $\mathrm{Constrain}^{\mathsf{R}}$ 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).

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

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

Figure 4: Dup and Drop instances

2.3 Type Class Instances

The key relation on predicates in our type system is 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 Γ^{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 [8] 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 Γ^{is}, which appears in figure 4.

The built-in instances specify restrictions on duplicating and dropping values, including values of compound type. 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 Ref^{rq} type. Then most of the table of restrictions in λ^{URAL} can be condensed into the two instances on the last line of figure 4.

2.4 Type Safety

Here we sketch part of the type safety proof; more details may be found in Gan's thesis [6].

The bulk of the work goes into proving preservation, and the key lemma in proving preservation is about relating constraints to locations. 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 Γ.

Lemma 1 (Constraints capture locations)**.** *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), and the remainder of the type safety proof is standard.

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 λ case. □

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 Jones's "Typing Haskell in Haskell" type checker [9]. 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 modifications to the Haskell type checker:

1. an elaboration pass that inserts dups and drops,
2. Dup and Drop type classes and instances, and
3. constraints on the free variables in λ terms, as in rule Abs.

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 at each program point.
- It imposes minimal type class constraints.

Due to space constraints we omit details of the proof of optimality here. The strategy is to recursively transform a term bottom-up, keeping track of the free variables fv in each recursively transformed subterm. Dup operations are inserted where the free variables of two subterms 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. To illustrate this, we describe the algorithm's behavior on linear pair ⊗ and with & respectively.

$$\begin{aligned}&\mathrm{infer}\,(e_1 \otimes e_2) = \mathbf{dup}\ \ \mathrm{fv}(e_1) \cap \mathrm{fv}(e_2)\ \ \mathbf{in}\ \ \mathrm{infer}(e_1) \otimes \mathrm{infer}(e_2)\\ &\mathrm{infer}\,(e_1 \,\&\, e_2) = (\mathbf{drop}\ \ \mathrm{fv}(e_2) \setminus \mathrm{fv}(e_1)\ \ \mathbf{in}\ \ \mathrm{infer}(e_1)) \,\&\, (\mathbf{drop}\ \ \mathrm{fv}(e_1) \setminus \mathrm{fv}(e_2)\ \ \mathbf{in}\ \ \mathrm{infer}(e_2))\end{aligned}$$

Establishing that this algorithm minimizes the set of free variables in each subterm is the key invariant.

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.

4 Related Work

Of the existing work in linear type systems, we will focus here on those which aim at general purpose linear types with polymorphism. The first linear type systems derive directly from intuitionistic linear logic, and use the exponential "!" to indicate types that support structural operations [1, 4]. Some later type systems, in order to support parametric polymorphism over linearity, replace "!" with types composed of a qualifier and a pretype [2, 16], 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 [3, 14]. 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 [13] and F° [10] 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 subkinding.

Compared to type systems like those of Clean and Alms, we believe Clamp offers advantages in simplicity and extensibility. Like Alms and F°, 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. 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 (§2.3). 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 Conclusion and Future Work

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 as well as a simple core theory and type checker. Type classes also make it easier to express polymorphism across the URAL lattice and encode 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 good base for substructural types.

In future work, we would like to allow programmers to define their own Dup and Drop instances, in order to specify custom resource management protocols. Additionally, we expect that extending the language of qualifiers in Clamp as in Clean or Alms is feasible and would give Clamp the qualifier polymorphism found in those languages.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1-2), April 1993.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *Proc. 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, 2005.

[3] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Math. Struct. Comp. Sci.*, 1996.

[4] G[avin] M. Bierman. *On Intuitionistic Linear Logic.* PhD thesis, University of Cambridge, August 1993.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82)*, 1982.

[6] Edward Gan. Clamp: Type classes for substructural types. Senior thesis, Harvard University, March 2013.

[7] Jean-Yves Girard. *Interprétation fonctionnelle et Elimination des coupures de l'arithmétique d'ordre supérieur.* These d'état, Université de Paris 7, 1972.

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

[9] Mark P. Jones. Typing Haskell in Haskell. In *Proc. 1999 Haskell Workshop*, 1999.

[10] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *Proc. 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2010.

[11] Simon Peyton Jones and John Hughes, ed. Haskell 98: A non-strict, purely functional language. February 1999.

[12] Jesse A. Tov. *Practical Programming with Substructural Types.* PhD thesis, Northeastern University, February 2012.

[13] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, 2011.

[14] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *Proc. 19th Annual Workshop on Implementation and Application of Functional Languages.* September 2007.

[15] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*, 1989.

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