Substructural Types

Justin Slepak

1 Three Structural Lemmas

Simply-typed λ-calculus follows three "structural" lemmas.

Lemma 1.1. Exchange:
If $\Gamma \vdash e : \tau$ *and* Γ' *is a permutation of* Γ*, then* $\Gamma' \vdash e : \tau$*.*

Lemma 1.2. Weakening:
If $\Gamma \vdash e : \tau$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : \tau' \vdash e : \tau$*.*

Lemma 1.3. Contraction:
If $(\Gamma_1, x_2 : \tau', x_3 : \tau', \Gamma_2) \vdash e : \tau$*, then* $(\Gamma_1, x_1 : \tau', \Gamma_2) \vdash e[x_1/x_2, x_1/x_3] : \tau$*.*

Exchange allows the environment to be rearranged without affecting typing judgments. Weakening means that the type environment can be extended with unused entries without affecting typing judgments. Contraction lets multiple environment entries with the same type be merged.

Substructural type systems do not uphold these properties. A *relevant* type system eliminates only weakening and requires that every variable be used at least once. An *affine* type system eliminates only contraction and prevents variables from being used more than once. A *linear* type system eliminates both weakening and contraction and mandates exactly one use of every variable. An *ordered* type system removes all three properties and requires variables to be used once each, in the order of their introduction.

2 λ^{URAL}

For practical purposes, it is more convenient to allow different variables to have different usage restrictions. ATTAPL [3] presents a language which allows both linear and unrestricted types. We will use λ^{URAL} [1], which allows unrestricted, relevant, affine, and linear types. We start with a simple core language based on simply-typed λ-calculus:

2.1 Syntax

$$
\begin{array}{rll}
e & ::= q\ n \mid q\lambda x : \tau.e \mid \texttt{if0}\ e\ e\ e \mid (e\ e) \mid (op\ e\ e) \mid x & \textit{(expressions)} \\
v & ::= q\ n \mid q\lambda x : \tau.e \mid x & \textit{(values)} \\
op & ::= + \mid - \mid * & \textit{(base operations)} \\
\tau & ::= q\ p & \textit{(types)} \\
p & ::= \texttt{Int} \mid \tau \to \tau & \textit{(pretypes)} \\
q & ::= \texttt{U} \mid \texttt{R} \mid \texttt{A} \mid \texttt{L} & \textit{(qualifiers)} \\
E & ::= E\ e \mid v\ E \mid (op\ E\ e) \mid (op\ v\ E) \mid (\texttt{if0}\ E\ e\ e) & \textit{(evaluation contexts)}
\end{array}
$$

2.2 Operational semantics

The operational semantics are essentially the same as for simply-typed λ-calculus. Certain terms, specifically λ-abstractions and numeric literals, must carry explicit type qualifiers. The only way λ-abstractions can be introduced (syntactically) necessitates specifying a qualifier, but $(op\ e\ e)$ introduces a new number without one, so we define it to have an unrestricted result:

$$(op\ (q_1\ n_1)\ (q_2\ n_2)) \mapsto (\mathtt{U}\ [\![n_1\ op\ n_2]\!])$$

2.3 Infrastructure for typing

A $\preceq$ order relation is needed to express the typing rules. It is defined first for pairs of qualifiers, and then lifted to operate on (type, qualifier) and (environment, qualifier) pairs. The relation on qualifiers is the reflexive-transitive closure of:

$$\mathtt{U} \preceq \mathtt{R}$$
$$\mathtt{U} \preceq \mathtt{A}$$
$$\mathtt{R} \preceq \mathtt{L}$$
$$\mathtt{A} \preceq \mathtt{L}$$

The relation is lifted for types by $(q\ \ p) \preceq q' \iff q \preceq q'$ and lifted for environments by $\Gamma \preceq q \iff \forall (x : q'\ p) \in \Gamma . q' \preceq q$, i.e. every entry in Γ must have a qualifier no more restrictive than q. The typing rules also rely on an "environment splitting" judgment, $\boxplus$.

$$\frac{}{\emptyset \boxplus \emptyset = \emptyset}$$

$$\frac{\Gamma = \Gamma_1 \boxplus \Gamma_2}{\Gamma, x : \tau = (\Gamma_1, x : \tau) \boxplus \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 \boxplus \Gamma_2}{\Gamma, x : \tau = \Gamma_1 \boxplus (\Gamma_2, x : \tau)}$$

$$\frac{\Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \tau \preceq \mathtt{R}}{\Gamma, x : \tau = (\Gamma_1, x : \tau) \boxplus (\Gamma_2, x : \tau)}$$

Intuitively, an environment can be split by partitioning into two separate environments. This does not introduce any weakening property, as every binding must appear on at least one side of the split. Depending on how a type derivation is examined, it may be useful to think of them as a "merge" operation: if a pair of environments can typecheck e_1 and e_2, their merge can typecheck $(e_1\ e_2)$. The merge of two environments is undefined if they share any linear or affine bindings, but relevant and unrestricted bindings may be shared by the two environments being merged. This introduces a limited form of contraction that only applies to relevant and unrestricted variables.

2.4 Typing

$$\frac{\Gamma_1 \vdash e : \tau \qquad \Gamma_2 \preceq \mathtt{A}}{\Gamma_1 \boxplus \Gamma_2 \vdash e : \tau}(\text{T-Weaken})$$

$$\frac{}{\bullet, x : \tau \vdash x : \tau}(\text{T-Var}) \qquad \frac{}{\bullet \vdash q\ n : q\ \mathtt{Int}}(\text{T-Int})$$

Typing a numeric literal requires the empty environment, and typing a variable requires an environment containing only that variable. The explicit weakening rule allows the empty or singleton environment to be expanded by adding bindings to match the larger environment that actually describes the current scope, but we can only add variables which are neither relevant nor linear (it also gives us the exchange property in the same way as TAPL's T-Var rule does). ATTAPL's presentation of the typing rules rolls the use of weakening into the variable and base-value literal rules.

$$\frac{\Gamma_1 \vdash e_{if} : q\ \mathtt{Int} \qquad \Gamma_2 \vdash e_{then} : \tau \qquad \Gamma_2 \vdash e_{else} : \tau}{\Gamma_1 \boxplus \Gamma_2 \vdash (\mathtt{if0}\ e_{if}\ e_{then}\ e_{else}) : \tau}(\text{T-If0})$$

Any affine and linear variables used in e_{if} are unavailable for use in e_{then} and e_{else}. Relevant and linear variables not used in e_{if} must be used in e_{then} and e_{else}. The environment splitting handles this arrangement.

$$\frac{\Gamma \preceq q \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash q\lambda x : \tau.e : q(\tau_1 \to \tau_2)}(\text{T-Abst})$$

A closure that references already-bound variables must respect their linear, affine, or relevant restrictions, so the same restrictions must apply to the closure itself. T-Weaken allows variables not referenced by this closure to be excluded from the environment used to typecheck it.

$$\frac{\Gamma_1 \vdash e_1 : q(\tau \to \tau') \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \boxplus \Gamma_2 \vdash (e_1\ e_2) : \tau'}(\text{T-App})$$

$$\frac{\Gamma_1 \vdash e_1 : q_1\mathtt{Int} \qquad \Gamma_2 \vdash e_2 : q_2\mathtt{Int}}{\Gamma_1 \boxplus \Gamma_2 \vdash (op\ e_1\ e_2) : \mathtt{U\ Int}}(\text{T-Arith})$$

The reasoning in T-App and T-Arith is similar to that for T-If0. Variables consumed by e_1 are unavailable for e_2. Arithmetic operators are defined to produce unrestricted values.

3 Language Extensions

We now extend the language with several new features, all of which have been seen in class as extensions for λ-calculus, but some complications are introduced by substructural typing.

3.1 Pairs and product types

$$e ::= \ldots \mid q < e, e > \mid \underline{\ ?\ } \qquad \textit{(expressions)}$$

$$v ::= \ldots \mid q < v, v > \qquad \textit{(values)}$$

$$p ::= \ldots \mid \tau \times \tau \qquad \textit{(pretypes)}$$

$$E ::= \ldots \mid q < E, e > \mid q < v, E > \qquad \textit{(evaluation contexts)}$$

The pair introduction form is obvious (simply add a qualifier); the immediate problem is how to construct an elimination form for pairs. Before, we used `fst` and `snd` operators to extract individual elements, but this keeps us from being able to access both terms of a linear or affine pair. Instead of `fst` and `snd`, we introduce a new binding form:

$$e ::= \ldots \mid \texttt{letpair}\ e < x, x > e \qquad \textit{(expressions)}$$

$$E ::= \ldots \mid \texttt{letpair}\ E < x, x > e \qquad \textit{(evaluation contexts)}$$

This form simultaneously binds both elements of a pair in another expression, so both elements can be accessed with a single use of the pair. We extend the operational semantics:

$$(\texttt{letpair}\ (q\ < v_1, v_2 >) < x_1, x_2 > e) \mapsto e[v_1/x_1, v_2/x_2]$$

Type checking is substantially similar to the structurally-typed λ-calculus, but we have a few restrictions on qualifiers. We should not allow, for example, an unrestricted pair of affine values, as the pair could be duplicated, bypassing the affine restrictions for its contents. So we require that a pair's qualifier be at least as restrictive as the qualifiers of its contents.

$$\frac{\Gamma_1 \vdash e_l : \tau_l \qquad \Gamma_2 \vdash e_r : \tau_r \qquad \tau_l \preceq q \qquad \tau_r \preceq q}{\Gamma_1 \boxplus \Gamma_2 \vdash (q < e_l, e_r >) : q\ (\tau_l \times \tau_r)} (\text{T-Pair})$$

Type checking a `letpair` looks similar to type checking a λ-abstraction, except two variables are introduced into the environment used to check the body.

$$\frac{\Gamma_1 \vdash e_{arg} : q\ (\tau_1 \times \tau_2) \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2 \vdash e_{body} : \tau}{\Gamma_1 \boxplus \Gamma_2 \vdash (\texttt{letpair}\ e_{arg} < x_1, x_2 > e_{body}) : \tau} (\text{T-LetPair})$$

3.2 Sum types

$$
\begin{array}{rll}
e & ::= \ldots \mid q\ \texttt{inL}_p\ e \mid q\ \texttt{inR}_p\ e & \textit{(expressions)} \\
& \quad \mid \texttt{case}\ e\ (\texttt{inL}\ x \Rightarrow e)\ (\texttt{inR}\ x \Rightarrow e) & \\
v & ::= \ldots \mid q\ \texttt{inL}_p\ v \mid q\ \texttt{inR}_p\ v & \textit{(values)} \\
p & ::= \ldots \mid \tau + \tau & \textit{(pretypes)} \\
E & ::= \ldots \mid q\ \texttt{inL}_p\ E \mid q\ \texttt{inR}_p\ E & \textit{(evaluation contexts)}
\end{array}
$$

The associated semantic extension is straightforward:

$$
\begin{array}{l}
(\texttt{case}\ (q\ \texttt{inL}_p\ v)\ (\texttt{inL}\ x_l \Rightarrow e_l)\ (\texttt{inR}\ x_r \Rightarrow e_r)) \mapsto e_l[v/x_l] \\
(\texttt{case}\ (q\ \texttt{inR}_p\ v)\ (\texttt{inL}\ x_l \Rightarrow e_l)\ (\texttt{inR}\ x_r \Rightarrow e_r)) \mapsto e_r[v/x_r]
\end{array}
$$

Again, typing a sum requires preventing usage restrictions from being bypassed by wrapping a value in a less restrictive sum type (but consider what happens if we modify our typing rules to allow something like $\texttt{U}\ (\texttt{U}\ p1) + (\texttt{L}\ p2)$ or $\texttt{U}\ (\texttt{L}\ p1) + (\texttt{L}\ p2)$ to be a well-formed type).

$$\frac{\Gamma \vdash e : \tau_l \qquad \tau_l \preceq q \qquad \tau_r \preceq q}{\Gamma \vdash (q\ \texttt{inL}_{\tau_l+\tau_r}\ e) : q\ (\tau_l + \tau_r)}(\text{T-INL})$$

$$\frac{\Gamma \vdash e : \tau_r \qquad \tau_l \preceq q \qquad \tau_r \preceq q}{\Gamma \vdash (q\ \texttt{inR}_{\tau_l+\tau_r}\ e) : q\ (\tau_l + \tau_r)}(\text{T-INR})$$

$$\frac{\Gamma_1 \vdash e : q\ (\tau_l + \tau_r) \qquad \Gamma_2, x_l : \tau_l \vdash e_l : \tau \qquad \Gamma_2, x_r : \tau_r \vdash e_r : \tau}{\Gamma_1 \boxplus \Gamma_2 \vdash (\texttt{case}\ e\ (\texttt{inL}\ x_l \Rightarrow e_l)\ (\texttt{inR}\ x_r \Rightarrow e_r)) : \tau}(\text{T-CASE})$$

3.3 Recursive types

$$
\begin{array}{rll}
e & ::= \ldots \mid \texttt{fold}_p\ e \mid \texttt{unfold}\ e & \textit{(expressions)} \\
v & ::= \ldots \mid \texttt{fold}_p\ v & \textit{(values)} \\
p & ::= \ldots \mid \alpha \mid \mu\alpha.\tau & \textit{(pretypes)} \\
E & ::= \ldots \mid \texttt{fold}_p\ E \mid \texttt{unfold}\ E & \textit{(evaluation contexts)}
\end{array}
$$

$$(\texttt{unfold}\ (\texttt{fold}_p\ v)) \mapsto v$$

Again, the concern is to prevent wrapping one type in a less restrictive type. We will let a `fold`ed expression keep the same qualifier as the pre-`fold` expression, i.e. the qualifier outside a $\mu\alpha.\tau$ pretype is ignored with use restrictions coming from τ's qualifier. Because of the addition of type variables, we must introduce a type variable environment, Δ, to check that types are well-formed.

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha] \qquad \bullet \vdash \tau[\mu\alpha.\tau/\alpha] \preceq q}{\Gamma \vdash (\texttt{fold}_{\mu\alpha.\tau}\ e) : q\ \mu\alpha.\tau}(\text{T-Fold})$$

$$\frac{\Gamma \vdash e : (q\ \mu\alpha.\tau)}{\Gamma \vdash (\texttt{unfold}\ e) : \tau[\mu\alpha.\tau/\alpha]}(\text{T-Unfold})$$

The typing rules for `fold` and `unfold` are straightforward and similar to the rules without substructural types. We cannot have type variables appear anywhere but a $\mu\alpha.\tau$ pretype, so we require that the empty environment be enough to prove our types to be well-formed. In proving this, it is necessary to alternate (mutually recursively) between proving well-formedness of a type and well-formedness of a pretype. We introduce this as a judgment that p is a well-formed pretype, $p : \texttt{P}$, or that τ is a well-formed type, $\tau : \texttt{T}$, but Δ contains only pretype variables. The rules for well-formedness are fairly intuitive, and there is only one way for variables to be introduced into the environment:

$$\frac{\Delta \vdash p : \texttt{P}}{\Delta \vdash (q\ p) : \texttt{T}} \qquad \frac{\Delta, \alpha \vdash \tau : \texttt{T}}{\Delta \vdash \mu\alpha.\tau : \texttt{T}} \qquad \frac{\Delta \vdash \tau : \texttt{T} \qquad \Delta \vdash \tau' : \texttt{T}}{\Delta \vdash q\ (\tau \to \tau') : \texttt{T}}$$

$$\frac{\Delta \vdash \tau : \texttt{T} \qquad \Delta \vdash \tau' : \texttt{T}}{\Delta \vdash q\ (\tau + \tau') : \texttt{T}} \qquad \frac{\Delta \vdash \tau : \texttt{T} \qquad \Delta \vdash \tau' : \texttt{T}}{\Delta \vdash q\ (\tau \times \tau') : \texttt{T}}$$

We also bring Δ into our judgments for $\preceq$. At this point, we needn't reference Δ when comparing qualifiers, but we use it in lifting $\preceq$ for types and environments.

3.4 Polymorphism

$$\begin{aligned}
e &::= \ldots \mid q\Lambda\alpha.e \mid e\langle p\rangle \mid q\Lambda\xi.e \mid e\langle q\rangle && \textit{(expressions)}\\
v &::= \ldots \mid q\Lambda\alpha.e \mid q\Lambda\xi.e && \textit{(values)}\\
p &::= \ldots \mid \forall\alpha.\tau \mid \forall\xi.\tau && \textit{(pretypes)}\\
q &::= \ldots \mid \xi && \textit{(qualifiers)}\\
E &::= \ldots \mid E\langle p\rangle \mid E\langle q\rangle && \textit{(evaluation contexts)}
\end{aligned}$$

We quantify separately over pretypes and qualifiers. This allows a quantified type to express that a function "works on `Int`s of any restriction level" (e.g. a purely arithmetic function) or "works for any unrestricted α" (e.g. a type signature for Church numerals) or "works on any types" (e.g. curry and uncurry). This syntax extension keeps α as the generic pretype variable and introduces ξ as the generic qualifier variable. The corresponding semantic extension follows:

$$(q\Lambda\alpha.e)\langle p\rangle \mapsto e[p/\alpha]$$
$$(q_0\Lambda\xi.e)\langle q_1\rangle \mapsto e[q_1/\xi]$$

Rather than requiring the programmer to explicitly specify a qualifier and pretype which will only be used with each other, we could even allow quantification over full types, introducing type variables and type application.

$$
\begin{array}{lll}
e & ::= \ldots \mid q\Lambda\alpha.e \mid e\langle p\rangle & \textit{(expressions)} \\
v & ::= \ldots \mid q\Lambda\alpha.e \mid q\Lambda\xi.e & \textit{(values)} \\
\tau & ::= \ldots \mid \sigma & \textit{(types)} \\
E & ::= \ldots \mid E\langle\tau\rangle & \textit{(evaluation contexts)}
\end{array}
$$

Checking pretype-polymorphic terms is similar to checking them in System F. Permitting qualifier polymorphism in addition to pretype polymorphism requires that our Δ also track qualifier variables (if pretype, qualifier, and type variables are to be drawn from the same set of identifiers, Δ must track $\alpha : \mathtt{P}$, $\xi : \mathtt{Q}$, and $\sigma : \mathtt{T}$ bindings, indicating whether a type-level variable represents a pretype, qualifier, or full type). At this point, we must also convert our old typing rules to use Δ in addition to Γ, as type-level variables may appear in expressions. We thread Δ through the old typing rules to ensure that types are well-formed, but none of the old rules are able to modify Δ (T-Fold must be changed to use Δ in place of $\bullet$). The introduction of qualifier variables means that we now need Δ to determine whether $\Gamma = \Gamma_1 \boxplus \Gamma_2$; our old typing rules must replace the $\Gamma_1 \boxplus \Gamma_2 \vdash e : \tau$ presentation by adding a premise $\Delta \vdash \Gamma = \Gamma_1 \boxplus \Gamma_2$ and rewriting the conclusion as $\Gamma \vdash e : \tau$. In order to handle qualifier variables in both new and old typing rules, we must also extend our $\preceq$ relation. We have $\mathtt{U}$ as the least restrictive qualifier and $\mathtt{L}$ as the most restrictive, so we can add $\mathtt{U} \preceq \xi$ and $\xi \preceq \mathtt{L}$ to the relation. However, knowing nothing more about ξ means we cannot extend the order any further.

$$\frac{\Gamma \preceq q \qquad \Delta, \alpha : \mathtt{P}; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash q\Lambda\alpha.e : q\forall\alpha.\tau}(\text{T-PreAbst})$$

$$\frac{\Delta; \Gamma \vdash e : q\forall\alpha.\tau \qquad \Delta \vdash p : \mathtt{P}}{\Delta; \Gamma \vdash e\langle p\rangle : \tau[p/\alpha]}(\text{T-PreApp})$$

$$\frac{\Gamma \preceq q \qquad \Delta, \xi : \mathtt{Q}; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash q\Lambda\xi.e : q\forall\xi.\tau}(\text{T-QualAbst})$$

$$\frac{\Delta; \Gamma \vdash e : q'\forall\xi.\tau \qquad \Delta \vdash q : \mathtt{Q}}{\Delta; \Gamma \vdash e\langle q\rangle : \tau[q/\xi]}(\text{T-QualApp})$$

$$\frac{\Gamma \preceq q \qquad \Delta, \sigma : \mathtt{T}; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash q\Lambda\sigma.e : q\forall\sigma.\tau}(\text{T-TypeAbst})$$

$$\frac{\Delta; \Gamma \vdash e : q\forall\alpha.\tau' \qquad \Delta \vdash \tau' : \mathtt{T}}{\Delta; \Gamma \vdash e\langle\tau\rangle : \tau'[\tau/\alpha]}(\text{T-TypeApp})$$

3.5 State

$$
\begin{array}{rll}
e & ::= \ldots \mid q\ \texttt{new}\ e \mid \texttt{free}\ e \mid \texttt{rd}\ e \mid \texttt{wr}\ e\ e \mid \texttt{sw}\ e\ e & \textit{(expressions)} \\
v & ::= \ldots \mid l \in \textsc{Locations} & \textit{(values)} \\
p & ::= \ldots \mid \texttt{ref}\ \tau & \textit{(pretypes)} \\
E & ::= \ldots \mid \texttt{new}\ q\ E \mid \texttt{free}\ E \mid \texttt{rd}\ E \mid \texttt{wr}\ E\ e \mid \texttt{wr}\ v\ E & \textit{(evaluation contexts)} \\
 & \quad \mid \texttt{sw}\ E\ e \mid \texttt{sw}\ v\ E &
\end{array}
$$

This extension places "locations," i.e. names of reference cells, which can be used alongside the store's variable mappings or in a store machine which simply operates on the control string and does not introduce variable mappings into the store.

The machine-oriented operational semantics for state maps a location to both its usage restrictions and its contents. The reason for having `sw`, a swap operation, in addition to `rd` and `wr` arises from typing restrictions and will be explained with the typing rules.

$$
\begin{aligned}
(S\ ;\ q\ \texttt{new}\ v) &\mapsto (S, l \Rightarrow (q\ v)\ ;\ l) \\
((S_1, l \Rightarrow (q\ v), S_2)\ ;\ \texttt{free}\ l) &\mapsto ((S_1, S_2)\ ;\ v) \\
((S_1, l \Rightarrow (q\ v), S_2)\ ;\ \texttt{rd}\ l) &\mapsto ((S_1, l \Rightarrow (q\ v), S_2)\ ;\ \texttt{L} < l, v >) \\
((S_1, l \Rightarrow (q\ v), S_2)\ ;\ \texttt{wr}\ l\ v') &\mapsto ((S_1, l \Rightarrow (q\ v'), S_2)\ ;\ l) \\
((S_1, l \Rightarrow (q\ v), S_2)\ ;\ \texttt{sw}\ l\ v') &\mapsto ((S_1, l \Rightarrow (q\ v'), S_2)\ ;\ \texttt{L} < l, v >)
\end{aligned}
$$

A reference cell that can only be accessed once is not useful, so accesses to a reference cell include a new instance of the reference in their return value. In the case of `rd` and `sw`, this necessitates constructing a pair so that the contents of the cell can also be returned. The pair is linear to avoid illicit duplication both of the reference and of the returned cell contents (recall that a pair type must be at least as restrictive as either of its component types). If either of these is legal to duplicate, that duplication can be performed after splitting the pair. It is important to remember that the q in a $q(\texttt{ref}\tau)$ refers not to the reference cell or the data it contains but to the reference itself. The τ behaves according to its own qualifier, and the reference cell remains in the heap until freed (explicitly or by a garbage collector). This brings up the question of what operations are compatible with what combinations of reference and data qualifiers.

$$
\frac{\Delta \vdash q \preceq A \qquad \Delta \vdash q' \preceq A \qquad \Delta; \Gamma \vdash e : q'\ p}{\Delta; \Gamma \vdash \texttt{new}\ q\ e : q\ (\texttt{ref}\ q'\ p)}(\text{T-NewUA})
$$

$$
\frac{\Delta \vdash \texttt{R} \preceq q \qquad \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash q\ \texttt{new}\ e : q\ (\texttt{ref}\ \tau)}(\text{T-NewRL})
$$

Separate typing rules are needed for constructing unrestricted/affine references and relevant/linear references. An unrestricted or affine reference cell may never be accessed, so it cannot contain relevant or linear data. On the other hand, a relevant or linear reference cell is guaranteed to be accessed, so it can contain anything (though the set of allowable operations will still depend on the qualifier of the cell contents). Notice that we are not confined according to the $\preceq$ lattice: it is legal for a relevant reference cell to contain affine or linear data or for an unrestricted cell to contain affine data.

$$\frac{\Delta;\Gamma \vdash e : q\ (\texttt{ref}\ \tau) \qquad \Delta \vdash \texttt{A} \preceq q}{\Delta;\Gamma \vdash \texttt{free}\ e : \tau}(\text{T-Free})$$

A relevant or unrestricted reference might still be used again later in the program, so freeing it is unsafe. Because this is a use of the reference, we can be sure that an affine or linear reference will not be used after it is freed.

$$\frac{\Delta;\Gamma \vdash e : q\ (\texttt{ref}\ \tau) \qquad \Delta \vdash \tau \preceq \texttt{R}}{\Delta;\Gamma \vdash \texttt{rd}\ e : \texttt{L}\ ((q\ \texttt{ref}\ \tau) \times \tau)}(\text{T-Read})$$

Multiple read references (possible due to the non-destructive nature of `rd`, as opposed to `free`) makes it possible to duplicate the contents of a reference cell. This is not permissible if the cell contains affine or linear data.

$$\frac{\begin{array}{c}\Delta \vdash \Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : q\ (\texttt{ref}\ \tau) \\ \Delta;\Gamma_2 \vdash e_2 : \tau \qquad \Delta \vdash \tau \preceq \texttt{A}\end{array}}{\Delta;\Gamma \vdash \texttt{wr}\ e_1\ e_2 : q\ \texttt{ref}\ \tau}(\text{T-WeakWrite})$$

The typing rule for weak (i.e. type-preserving) writes must ensure compatibility between the reference and contents expressions, i.e. that the program is writing a τ into a $\texttt{ref}\ \tau$. It must also ensure that the program is not overwriting a relevant or linear value because that would allow such values to be destroyed without ever being used.

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : q\ (\texttt{ref}\ \tau) \qquad \Delta;\Gamma_2 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \texttt{sw}\ e_1\ e_2 : \texttt{L}\ ((q\ \texttt{ref}\ \tau) \times \tau)}(\text{T-WeakSwap})$$

What can we do with reference cells that contain linear data? Reading requires the contents to be no more restricted than `R`. Writing requires the contents to be no more restricted than `A`. Tracking whether a cell has been referenced already in order to decide whether to allow reading/writing is too complicated. Instead, we require that the cell contents always keep a "not-yet-used" status. Then we can take a linear (or less restricted) value out of a cell by exchanging it with another value of the same type. This allows us to write the cell contents without the $\tau \preceq \texttt{A}$ requirement and to read the cell contents without the $\tau \preceq \texttt{R}$ requirement.

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e_1 : q\ (\texttt{ref}\ \tau) \qquad \Delta; \Gamma_2 \vdash e_2 : \tau \qquad \Delta \vdash \tau \preceq \texttt{A} \qquad \Delta \vdash \texttt{A} \preceq q \qquad \Delta \vdash \tau' \preceq q}{\Delta; \Gamma \vdash \texttt{wr}\ e_1\ e_2 : q\ \texttt{ref}\ \tau} \text{(T-StrongWrite)}$$

In order to make a strong (i.e. type-altering) write, we must be certain that the reference in question is unique (i.e. affine or linear) so that typechecking other parts of the program does not need to consider whether this reference has changed its type. We also require that the new content type be permissive enough to be allowed in this reference (e.g. a relevant reference cannot be updated to contain affine data, but a linear reference can).

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e_1 : q\ (\texttt{ref}\ \tau) \qquad \Delta; \Gamma_2 \vdash e_2 : \tau \qquad \Delta \vdash \texttt{A} \preceq q \qquad \Delta \vdash \tau' \preceq q}{\Delta; \Gamma \vdash \texttt{sw}\ e_1\ e_2 : \texttt{L}\ ((q\ \texttt{ref}\ \tau) \times \tau)} \text{(T-StrongSwap)}$$

The same restrictions added for strong writes also apply to strong swaps.

4 Progress and preservation for λ^{UAL}

Our type system is meant to ensure certain values are not used multiple times. In order to state progress and preservation for this system in such terms, we must express the operational semantics in terms of a machine with a variable store. To show how the store is updated by accesses to affine and linear variables, we must define a $\sim_q$ operation:

$$S \sim_U x = S$$
$$(S_1, x \mapsto v, S_2) \sim_A x = (S_1, S_2)$$
$$(S_1, x \mapsto v, S_2) \sim_L x = (S_1, S_2)$$

This removes affine/linear entries from the store upon use. The machine's strategy will be to select an evaluation context and then evaluate the expression in its hole.

$$\frac{(S; e) \mapsto_\beta (S'; e')}{(S; E[e]) \mapsto (S'; E[e'])}$$

The machine-oriented semantics reduces expressions to variables, which refer to data in the store. Not all of the rules are given here, as they are straightforward transformations from the expression-only semantics. The general theme is that steps which compute new values create new store entries with fresh names.

$$(S\ ;\ q\ n) \mapsto_\beta (S, x \mapsto q\ n\ ;\ x),$$
with fresh x

$$((S_1, x \mapsto q\ 0, S_2); (\texttt{if0}\ x\ e_1\ e_2)) \mapsto_\beta ((S_1, x \mapsto q\ 0, S_2) \sim_q x\ ;\ e_1)$$

$$((S_1, x \mapsto q\ n, S_2)\ ;\ (\texttt{if0}\ x\ e_1\ e_2)) \mapsto_\beta ((S_1, x \mapsto q\ 0, S_2) \sim_q x\ ;\ e_2),$$
where $n \neq 0$

$$(S\ ;\ q\lambda x_1 : \tau.e) \mapsto_\beta (S, x_0 \mapsto q\lambda x_1 : \tau.e\ ;\ x_0),$$
with fresh x_0

$$((S_1, x_0 \mapsto q\lambda x_1 : \tau.e), S_2)\ ;\ (x_0\ x_2)) \mapsto_\beta ((S_1, x_0 \mapsto q\lambda x_1 : \tau.e, S_2) \sim_q x_0\ ; e[x_2/x_1])$$

$$(S\ ;\ (q < x_l, x_r >)) \mapsto_\beta ((S, x_0 \mapsto (q < x_l, x_r >))\ ;\ x_0),$$
with fresh x_0

etc.

We also require a notion of store typing. In essense, we map a store to an environment, which should then be capable of typechecking the expression the machine has at the same time as it has that store. Thus, already-used affine and linear variables are excluded from the environment.

$$\frac{}{\vdash \emptyset : \emptyset} \qquad \frac{\vdash S : \Gamma \boxplus \Gamma' \qquad \Gamma' \vdash x : \tau}{\vdash S, x \mapsto v : \Gamma, x : \tau}$$

Finally, we combine our rules in order to judge well-typedness for machine states, i.e. store/expression pairs:

$$\frac{\vdash S : \Gamma \qquad \Gamma \vdash e : \tau}{\vdash (S\ ;\ e)}$$

This gives us the machinery we need to state progress and preservation.

Theorem 4.1. Progress:
If $\vdash (S\ ;\ e)$*, then* $\exists S', e'.(S\ ;\ e) \mapsto (S'\ ;\ e')$ *or* e *is a value.*

Theorem 4.2. Preservation:
If $\vdash (S\ ;\ e)$ *and* $(S\ ;\ e) \mapsto (S'\ ;\ e')$*, then* $\vdash (S'\ ;\ e')$.

But what about relevant types? Our machine knows exactly when to eliminate linear and affine variables, but removing a relevant variable is unsafe as it may be used again in the future (i.e. progress would be lost). We also cannot simply leave it there because the machine will eventually reach a state where the corresponding Γ will fail to typecheck its control expression (i.e. preservation would be lost). We can introduce a "used" flag into each store entry. On creation, the flag is clear, and it is set when the variable is used by the control string redex (this flag only needs to be managed for relevant variables). The environment we get from store typing would also track whether relevant entries are already-used, and they can be considered as subject to T-Weaken.

5 Algorithmic type checking for λ^{UAL}

In order to derive $\Gamma \vdash e : \tau$, the type checker must often choose how to split Γ into Γ_1 and Γ_2. If a mechanical type checker failes to prove $\Gamma \vdash e : \tau$, then it must backtrack and try again with different Γ_1 and Γ_2 until it knows that no choice of Γ_1 and Γ_2 will work. To avoid nondeterminism in type checking, we introduce the "output environment" associated with a typing judgment: $\Gamma \vdash e : \tau \mathbin{//} \Gamma'$. Γ' contains all entries from Γ not "used up" in deriving $e : \tau$. We then replace $\boxplus$, the "environment splitting" operation, with $\div$, a deterministic "environment difference" operation:

$$\frac{}{\Gamma \div \emptyset = \Gamma}$$

$$\frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \qquad (x : \tau) \notin \Gamma_3 \qquad \mathtt{A} \preceq \tau}{\Gamma_1 \div (\Gamma_2, x : \tau) = \Gamma_3}$$

$$\frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \qquad \Gamma_3 = \Gamma_4, x : \tau, \Gamma_5 \qquad \tau \preceq \mathtt{U}}{\Gamma_1 \div (\Gamma_2, x : \tau) = \Gamma_4, \Gamma_5}$$

We use this operation to remove variables at the end of their scope, and we ensure at removal that the variables are not affine or linear variables left untouched during their scope. We accept a term as well-typed if the output environment contains no linear variables.

$$\frac{}{\Gamma \vdash q\ n : q\ \mathtt{Int} \mathbin{//} \Gamma}(\text{A-Int})$$

$$\frac{\tau \preceq \mathtt{U}}{\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau \mathbin{/} \Gamma_1, x : \tau, \Gamma_2}(\text{A-UVar})$$

$$\frac{\mathtt{A} \preceq \tau}{\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau \mathbin{//} \Gamma_1, x : \tau, \Gamma_2}(\text{A-ALVar})$$

Using an integer or unrestricted variable does not alter the environment, whereas using a linear variable requires that it be removed from the output environment.

$$\frac{\Gamma_1 \vdash e_1 : \tau \to \tau' \mathbin{//} \Gamma_2 \qquad \Gamma_2 \vdash e_2 : \tau \mathbin{//} \Gamma_3}{\Gamma_1 \vdash (e_1\ e_2) : \tau' \mathbin{//} \Gamma_3}(\text{A-App})$$

$$\frac{\Gamma_1 \vdash e_1 : q\ \mathtt{Int} \mathbin{//} \Gamma_2 \qquad \Gamma_2 \vdash e_2 : q'\ \mathtt{Int} \mathbin{//} \Gamma_3}{\Gamma_1 \vdash (op\ e_1\ e_2) : \mathtt{U}\ \mathtt{Int} \mathbin{//} \Gamma_3}(\text{A-Arith})$$

Checking multiple subexpressions chains the environment changes together. Variables consumed by e_1 are unavailable for e_2. Variables consumed by either e_1 or e_2 are considered consumed by $(e_1\ e_2)$ or $(op\ e_1\ e_2)$.

$$\frac{\Gamma_1, x : \tau \vdash e : \tau' \mathbin{//} \Gamma_2 \qquad q \preceq \mathtt{U} \Rightarrow \Gamma_2 \div (x : \tau) = \Gamma_1}{\Gamma_1 \vdash q\lambda x : \tau.e : q\ (\tau \to \tau') \mathbin{//} \Gamma_2 \div (x : \tau)}(\text{A-Abst})$$

Giving $\Gamma_2 \div (x : \tau)$ as the output environment for typechecking a λ-abstraction removes the parameter variable from the environment at the end of its scope. In the case of an unrestricted closure, we use the $\Gamma_2 \div (x : \tau) = \Gamma_1$ requirement to ensure the closure does not use any linear variables from its environment. If it does, Γ_2 will have them removed, and $\Gamma_2 \div (x : \tau)$ will exclude those bindings.

$$\frac{\Gamma_1 \vdash e_{if} : q\ \mathtt{Int} \mathbin{//} \Gamma_2 \qquad \Gamma_2 \vdash e_{then} : \tau \mathbin{//} \Gamma_3 \qquad \Gamma_2 \vdash e_{else} : \tau \mathbin{//} \Gamma_3}{\Gamma_1 \vdash (\mathtt{if0}\ e_{if}\ e_{then}\ e_{else}) : \tau \mathbin{//} \Gamma_3}(\text{A-If0})$$

This rule is similar to T-App and T-Arith. By mandating the same output environment when checking e_{then} and e_{else}, we require that they comsume the same linear variables.

References

[1] Amal Ahmed. A step-indexed model of substructural state. In *In: Proc. International Conference on Functional Programming. (2005) 7891*, pages 78–91. ACM Press, 2005.

> This paper describes λ^{URAL}. The implications of substructural reference types are the most important insight for this lecture. Included is a table describing what data may be stored in and operations may be performed on reference cells according to their qualifiers. Instead of progress and preservation, type soundness is proven via a step-indexed logical relation.

[2] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L 3 : A linear language with locations. In *In Seventh International Conference on Typed Lambda Calculi and Applications*, pages 293–307, 2005.

> This paper describes a language which uses substructural state to allow strong updates of reference cells. Because a linear reference cell is only accessible from one part of the program at a time, the type checker is able to track type-changing updates to the cell.

[3] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

> Chapter 1 covers substructural types, with an emphasis on a type system which includes linear and unrestricted types (but not affine or relevant types). Topics not covered in this lecture include: ordered types, which eliminate all three substructural lemmas mentioned at the beginning; reference-counted types, which introduce a controlled way to duplicate affine or linear references; a language system which uses affine types to control the time a function may expend while running; and compiler optimizations which can take advantage of the information carried by type qualifiers.