2 *Substructural Type Systems*

By David Walker

Advanced type systems make it possible to restrict access to data structures and to limit the use of newly-defined operations. Oftentimes, this sort of access control is achieved through the definition of new abstract types under control of a particular module. For example, consider the following simplified file system interface.

```
type file

val open   : string → file option
val read   : file → string * file
val append : file * string → file
val write  : file * string → file
val close  : file → unit
```

By declaring that the type `file` is abstract, the implementer of the module can maintain strict control over the representation of files. A client has no way to accidentally (or maliciously) alter any of the file's representation invariants. Consequently, the implementer may assume that the invariants that he or she establishes upon opening a file hold before any `read`, `append`, `write` or `close`.

While abstract types are a powerful means of controlling the structure of data, they are not sufficient to limit the *ordering* and *number of uses* of functions in an interface. Try as we might, there is no (static) way to prevent a file from being read after it has been closed. Likewise, we cannot stop a client from closing a file twice or forgetting to close a file.

This chapter introduces *substructural* type systems that augment standard type abstraction mechanisms with the ability to control the number and order of uses of a data structure or operation. Substructural type systems are particularly useful for constraining interfaces that provide access to system resources such as files, locks and memory. Each of these resources undergoes a series of changes of state throughout its lifetime. Files, as we have seen, may be open or closed; locks may be held or not; and memory may be allocated or deallocated. Substructural type systems provide sound static mechanisms for keeping track of just these sorts of state changes and prevent operations on objects in an invalid state.

The bulk of this chapter will focus on applications of substructural type systems to the control of memory resources. Memory is a pervasive resource that must be managed carefully in any programming system so it makes an excellent target of study. However, the general principles that we establish can be applied to other sorts of resources as well.

2.1 Structural Properties

All of the type systems we have seen so far in this book allow *unrestricted* use of variables in the type checking context. For instance, each variable may be used once, twice, three times, or not at all. It turns out that this is not just an idle observation: A precise analysis of the properties of such variables will suggest a whole new collection of type systems.

To begin our exploration, we will analyze the simply-typed lambda calculus, which is reviewed in Figure 2-1. In this discussion, we are going to be particularly careful when it comes to the form of the type-checking context Γ. We will consider such contexts to be simple lists of variable-type pairs. The "," operator appends a pair to the end of the list. We also write (Γ_1, Γ_2) for the list that results from appending Γ_2 onto the end of Γ_1. As usual, we allow a given variable to appear at most once in a context and to maintain this invariant, we implicitly alpha-convert bound variables before entering them into the context.

We are now in position to consider three basic *structural* properties satisfied by our simply-typed lambda calculus. The first property, *exchange*, indicates that the order in which we write down variables in the context is irrelevant. A corollary of exchange is that if we can type check a term with the context Γ, then we can type check that term with any permutation of the variables in Γ. The second property, *weakening*, indicates that adding extra, unneeded assumptions to the context, does not prevent a term from type checking. Finally, the third property, *contraction*, states that if we can type check a term using two identical assumptions ($\mathtt{x}_2\mathtt{:T}_1$ and $\mathtt{x}_3\mathtt{:T}_1$) then we can check the same term using a single assumption.

2.1.1 LEMMA [EXCHANGE]: If $\Gamma_1, \mathtt{x}_1\mathtt{:T}_1, \mathtt{x}_2\mathtt{:T}_2, \Gamma_2 \vdash \mathtt{t : T}$ then

Syntax

`b` ::=		*booleans:*
	`true`	*true*
	`false`	*false*
`t` ::=		*terms:*
	`x`	*variable*
	`b`	*boolean*
	`if t then t else t`	*conditional*
	λ`x:T.t`	*abstraction*
	`t t`	*application*
`T` ::=		*types:*
	`Bool`	*booleans*
	`T`$\to$`T`	*type of functions*
Γ ::=		*contexts:*
	$\emptyset$	*empty context*
	Γ, `x:T`	*term variable binding*

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{}{\Gamma_1, \mathtt{x}:\mathtt{T}, \Gamma_2 \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-VAR)}$$

$$\frac{}{\Gamma \vdash \mathtt{b} : \mathtt{Bool}} \quad \text{(T-BOOL)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t3} : \mathtt{T}} \quad \text{(T-IF)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-APP)}$$

Figure 2-1: Simply-typed lambda calculus with booleans

$\Gamma_1, \mathtt{x}_2:\mathtt{T}_2, \mathtt{x}_1:\mathtt{T}_1, \Gamma_2 \vdash \mathtt{t} : \mathtt{T}$ □

2.1.2 LEMMA [WEAKENING]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ then $\Gamma, \mathtt{x}_1:\mathtt{T}_1 \vdash \mathtt{t} : \mathtt{T}$ □

2.1.3 LEMMA [CONTRACTION]: If $\Gamma, \mathtt{x}_2:\mathtt{T}_1, \mathtt{x}_3:\mathtt{T}_1 \vdash \mathtt{t} : \mathtt{T}_3$ then
$\Gamma, \mathtt{x}_1:\mathtt{T}_1 \vdash [\mathtt{x}_2 \mapsto \mathtt{x}_1][\mathtt{x}_3 \mapsto \mathtt{x}_1]\mathtt{t} : \mathtt{T}_3$ □

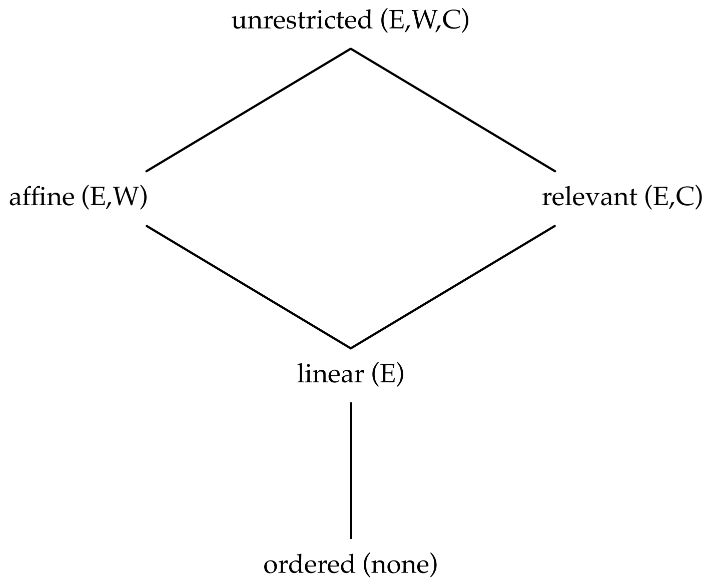
2.1.4 EXERCISE [★,RECOMMENDED]: Prove exchange, weakening and contraction hold for the simply-typed lambda calculus. □

A *substructural type system* is any type system that is designed so that one or more of the structural properties do not hold. Different substructural type systems arise when different properties are withheld.

- *Linear* type systems ensure that every variable is used exactly once by allowing exchange but not weakening or contraction.
- *Affine* type systems ensure that every variable is used at most once by allowing exchange and weakening, but not contraction.
- *Relevant* type systems ensure that every variable is used at least once by allowing exchange and contraction, but not weakening.

- *Ordered* type systems ensure that every variable is used exactly once and is used in the order in which it is introduced. Ordered type systems do not allow any of the structural properties.

The picture below can serve as a mnemonic for the relationship between these systems. The system at the bottom of the diagram (the ordered type system) admits no structural properties. As we proceed upwards in the diagram, we add structural properties: E stands for exchange; W stands for weakening; and C stands for contraction. It might be possible to define type systems containing other combinations of structural properties, such as contraction only or weakening only, but so far researchers have not found applications for such combinations. Consequently, we have excluded them from the diagram.



The diagram can be realized as a relation between the systems. We say system q_1 is more restrictive than system q_2 and write $\mathsf{q}_1 \sqsubseteq \mathsf{q}_2$ when system q_1 exhibits fewer structural rules than system q_2. Figure 2-2 specifies the relation, which we will find useful in the coming sections of this chapter.

`q ::=`		*system:*
	`ord`	*ordered*
	`lin`	*linear*
	`rel`	*relevant*
	`aff`	*affine*
	`un`	*unrestricted*

$$\texttt{ord} \sqsubseteq \texttt{lin} \qquad \text{(Q-OrdLin)}$$

$$\texttt{lin} \sqsubseteq \texttt{rel} \qquad \text{(Q-LinRel)}$$

$$\texttt{lin} \sqsubseteq \texttt{aff} \qquad \text{(Q-LinAff)}$$

$$\texttt{rel} \sqsubseteq \texttt{un} \qquad \text{(Q-RelUn)}$$

$$\texttt{aff} \sqsubseteq \texttt{un} \qquad \text{(Q-AffUn)}$$

$$\texttt{q} \sqsubseteq \texttt{q} \qquad \text{(Q-Reflex)}$$

$$\frac{\texttt{q}_1 \sqsubseteq \texttt{q}_2 \qquad \texttt{q}_2 \sqsubseteq \texttt{q}_3}{\texttt{q}_1 \sqsubseteq \texttt{q}_3} \qquad \text{(Q-Trans)}$$

Figure 2-2: A Relation between Substructural Type Systems

2.2 A Linear Type System

In order to safely deallocate data, we need to know that the data we deallocate is never used in the future. Unfortunately, we cannot, in general, deduce whether data will be used after execution passes a certain program point: The problem is clearly undecidable. However, there are a number of sound, but useful approximate solutions. One such solution may be implemented using a *linear type system*. Linear type systems ensure that objects are used exactly once, so it is completely obvious that after the use of an object, it may be safely deallocated.

Syntax

Figure 2-3 presents the syntax of our linear language, which is an extension of the simply-typed lambda calculus. The main addition to be aware of, at this point, are the type qualifiers `q` that annotate the introduction forms for all data structures. The linear qualifier (`lin`) indicates that the data structure in question will be *used* (i.e., appear in the appropriate elimination form) exactly once in the program. Operationally, we deallocate these linear values immediately after they are used. The unrestricted qualifier (`un`) indicates that the data structure behaves as in the standard simply-typed lambda calculus. In other words, unrestricted data can be used as many times as desired and its memory resources will be automatically recycled by some extra-linguistic mechanism (a conventional garbage collector).

Apart from the qualifiers, the only slightly unusual syntactic form is the

Syntax

`q`	::=		*qualifiers:*
		`lin`	*linear*
		`un`	*unrestricted*
`b`	::=		*booleans:*
		`true`	*true*
		`false`	*false*
`t`	::=		*terms:*
		`x`	*variable*
		`q b`	*boolean*
		`if t then t else t`	*conditional*
		`q <t,t>`	*pair*
		`split t as x,y in t`	*split*
		`q` λ`x:T.t`	*abstraction*
		`t t`	*application*
`P`	::=		*pretypes:*
		`Bool`	*booleans*
		`T*T`	*pairs*
		`T`$\rightarrow$`T`	*functions*
`T`	::=		*types:*
		`q P`	*qualified pretype*
Γ	::=		*contexts:*
		$\emptyset$	*empty context*
		Γ`, x:T`	*term variable binding*

Figure 2-3: Linear lambda calculus: Syntax

elimination form for pairs. The term `split` t_1 `as x,y in` t_2 projects the first and second components from the pair t_1 and calls them `x` and `y` in t_2. This `split` operation allows us to extract two components while only counting a single use of a pair.[1]

To avoid dealing with an unnecessarily heavy syntax, we adopt a couple abbreviations in our examples in this section. First, we omit all unrestricted qualifiers and only annotate programs with the linear ones. Second, we freely use n-ary tuples (triples, quadruples, unit, etc.) in addition to pairs and also allow multi-argument functions. The latter may be defined as single-argument functions that take linear pairs (triples, etc) as arguments and immediately split them upon entry to the function body. Third, we often use ML-style type declarations, value declarations and let expressions where convenient; they all have the obvious meanings.

Typing

In order to ensure that linear objects are used exactly once, our type system maintains two important invariants.

1. Linear variables are used exactly once along every control-flow path.

1. Extracting two components using the more conventional projections π_1 t_1 and π_2 t_1 requires two uses of the pair t_1. While it is possible to arrange our language so that operations of this sort are available, the set-up is somewhat trickier, so we simply avoid it here.

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\emptyset = \emptyset \circ \emptyset \quad \text{(M-Empty)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:un P} = (\Gamma_1, \texttt{x:un P}) \circ (\Gamma_2, \texttt{x:un P})} \quad \text{(M-Un)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:lin P} = (\Gamma_1, \texttt{x:lin P}) \circ \Gamma_2} \quad \text{(M-Lin1)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:lin P} = \Gamma_1 \circ (\Gamma_2, \texttt{x:lin P})} \quad \text{(M-Lin2)}$$

Figure 2-4: Linear Context Splitting Rules

2. Unrestricted data structures may not contain linear data structures. More generally, data structures with less restrictive type may not contain data structures with more restrictive type.

To understand why these invariants are useful, consider what could happen if either invariant is broken. When considering the first invariant, assume we have constructed a function `use` that makes some use of its argument before deallocating it. Now, if we allow a linear variable (say `x`) to appear twice, a programmer might write `<use x,use x>`, or, slightly more deviously, `(λz.λy.<use z,use y>) x x`. In either case, the program ends up "using" `x` twice, causing the program to crash.

Now considered the second invariant and suppose we allow a linear data structure (call it `x`) to appear inside an unrestricted pair (`un <x,3>`). We can get exactly the same effect as above by using the unrestricted data structure multiple times:

```
let z = un <x,3> in
split z as x1,_ in
split z as x2,_ in
<use x1,use x2>
```

Fortunately, our type system ensures that none of these situations can occur.

We maintain the first invariant through careful context management. When type checking terms with two or more subterms, we pass all of the unrestricted variables in the context to each subterm. However, we split the linear variables between the different subterms in such a way as to ensure each variable is used exactly once. Figure 2-4 defines a relation, $\Gamma = \Gamma_1 \circ \Gamma_2$, which describes how to split a single context in a rule conclusion (Γ) into two contexts (Γ_1 and Γ_2) that will be used to type different subterms in a rule premise.

To help check the second invariant, we define the predicate `q(T)` (and its extension to contexts `q(`Γ`)`) to express the types `T` that can appear in a `q`-

qualified data structure. These containment rules state that linear data structures can hold objects with linear or unrestricted type, but unrestricted data structures can only hold objects with unrestricted type.

- $\mathtt{q(T)}$ if and only if $\mathtt{T} = \mathtt{q'\ P}$ and $\mathtt{q} \sqsubseteq \mathtt{q'}$
- $\mathtt{q}(\Gamma)$ if and only if $(\mathtt{x{:}T}) \in \Gamma$ implies $\mathtt{q(T)}$

Recall, we have already defined $\mathtt{q} \sqsubseteq \mathtt{q'}$ such that it is reflexive, transitive and $\mathtt{lin} \sqsubseteq \mathtt{un}$.

Now that we have defined the rules for containment and context splitting, we are ready for the typing rules proper, which appear in Figure 2-5. Keep in mind that these rules are constructed anticipating a call-by-value operational semantics.

It is often the case when designing a type system that the rules for the base cases, variables and constants, are hardly worth mentioning. However, in substructural type systems these base cases have a special role in defining the nature of the type system, and subtle changes can make all the difference. In our linear system, the base cases must ensure that no linear variable is discarded without being used. To enforce this invariant in rule (T-VAR), we explicitly check that Γ_1 and Γ_2 contain no linear variables using the condition $\mathtt{un}\ (\Gamma_1, \Gamma_2)$. We make a similar check in rule (T-BOOL). Notice also that rule (T-VAR) is written carefully to allow the variable $\mathtt{x}$ to appear anywhere in the context, rather than just at the beginning or at the end.

2.2.1 EXERCISE [★]: What is the effect of rewriting the variable rule as follows?

$$\frac{\mathtt{un}\ (\Gamma)}{\Gamma, \mathtt{x{:}T} \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-BROKENVAR)}$$

□

The inductive cases of the typing relation take care to use context splitting to partition linear variables between various subterms. For instance, rule (T-IF) splits the incoming context into two parts, one which is used to check subterm $\mathtt{t}_1$ and the other which is used to check both $\mathtt{t}_2$ and $\mathtt{t}_3$. As a result, a particular linear variable will occur once in $\mathtt{t}_2$ and once in $\mathtt{t}_3$. However, the linear object bound to the variable in question will be used (and hence deallocated) exactly once at run time since only one of $\mathtt{t}_2$ or $\mathtt{t}_3$ will be executed.

The rules for creation of pairs and functions make use of the containment rules. In each case, the data structure's qualifier $\mathtt{q}$ is used in the premise of the typing rule to limit the sorts of objects it may contain. For example, in the rule (T-ABS), if the qualifier $\mathtt{q}$ is $\mathtt{un}$ then the variables in Γ, which will

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathrm{un}\ (\Gamma_1, \Gamma_2)}{\Gamma_1, \mathtt{x}{:}\mathtt{T}, \Gamma_2 \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-VAR)}$$

$$\frac{\mathrm{un}\ (\Gamma)}{\Gamma \vdash \mathtt{q\ b} : \mathtt{q\ Bool}} \quad \text{(T-BOOL)}$$

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{q\ Bool} \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma_2 \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \quad \text{(T-IF)}$$

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \mathtt{q(T_1)} \qquad \mathtt{q(T_2)}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{q} \mathtt{<t_1,t_2>} : \mathtt{q\ (T_1{*}T_2)}} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{q\ (T_1{*}T_2)} \qquad \Gamma_2, \mathtt{x}{:}\mathtt{T}_1, \mathtt{y}{:}\mathtt{T}_2 \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{split}\ \mathtt{t}_1\ \mathtt{as}\ \mathtt{x,y}\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}} \quad \text{(T-SPLIT)}$$

$$\frac{\mathtt{q}(\Gamma) \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{q}\ \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{q}\ \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{q}\ \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-APP)}$$

Figure 2-5: Linear lambda calculus: Typing

inhabit the function closure, must satisfy un (Γ). In other words, they must all have unrestricted type. If we omitted this constraint, we could write the following badly behaved functions.[2]

```
type T = un (un bool → lin bool)

val discard =
  lin λx:lin bool.
    (λf:T.lin true) (un λy:un bool.x)

val duplicate =
  lin λx:lin bool.
    (λf:T.lin <f (un true),f (un true)>)) (un λy:un bool.x)
```

The first function discards a linear argument `x` without using it and the second duplicates a linear argument and returns two copies of it in a pair. Hence, in the first case, we fail to deallocate `x` and in the second case, a subsequent function may project both elements of the pair and use `x` twice, which would result in a memory error as `x` would be deallocated immediately after the first use. Fortunately, the containment constraint disallows the linear variable `x` from appearing in the unrestricted function `(λy:bool. x)`.

2. For clarity, we have not omitted the unrestricted qualifiers in this example.

Now that we have defined our type system, we should verify our intended structural properties: exchange for all variables, and weakening and contraction for unrestricted variables.

2.2.2 LEMMA [EXCHANGE]: If $\Gamma_1, \mathtt{x}_1 \mathtt{:T}_1, \mathtt{x}_2 \mathtt{:T}_2, \Gamma_2 \vdash \mathtt{t : T}$ then $\Gamma_1, \mathtt{x}_2 \mathtt{:T}_2, \mathtt{x}_1 \mathtt{:T}_1, \Gamma_2 \vdash \mathtt{t : T}$. □

2.2.3 LEMMA [UNRESTRICTED WEAKENING]: If $\Gamma \vdash \mathtt{t : T}$ then $\Gamma, \mathtt{x}_1 \mathtt{:un\ P}_1 \vdash \mathtt{t : T}$. □

2.2.4 LEMMA [UNRESTRICTED CONTRACTION]:
If $\Gamma, \mathtt{x}_2 \mathtt{:un\ P}_1, \mathtt{x}_3 \mathtt{:un\ P}_1 \vdash \mathtt{t : T}_3$ then
$\Gamma, \mathtt{x}_1 \mathtt{:un\ P}_1 \vdash [\mathtt{x}_2 \mapsto \mathtt{x}_1][\mathtt{x}_3 \mapsto \mathtt{x}_1]\mathtt{t : T}_3$. □

Proof: The proofs of all three lemmas follow by induction on the structure of the appropriate typing derivation. □

Algorithmic Linear Type Checking

The inference rules provided in the previous subsection give a clear, concise specification of the linearly-typed programs. However, these rules are also highly non-deterministic and cannot be implemented directly. The primary difficulty is that to implement the non-deterministic splitting operation, $\Gamma = \Gamma_1 \circ \Gamma_2$, we must guess how to split an input context Γ into two parts. Fortunately, it is relatively straightforward to restructure the type checking rules to avoid having to make these guesses. This restructuring leads directly to a practical type checking algorithm.

The central idea is that rather than splitting the context into parts before checking a complex expression composed of several subexpressions, we can pass the entire context as an input to the first subexpression and have it return the unused portion as an output. This output may then be used to check the next subexpression, which may also return some unused portions of the context as an output, and so on. Figure 2-6 makes these ideas concrete. It defines a new algorithmic type checking judgment with the form $\Gamma_{in} \vdash \mathtt{t : T;}\Gamma_{out}$, where Γ_{in} is the input context, some portion of which will be consumed during type checking of $\mathtt{t}$, and Γ_{out} is the output context, which will be synthesized alongside the type $\mathtt{T}$.

There are several key changes in our reformulated system. First, the base cases for variables and constants allow any context to pass through the judgment rather than restricting the number of linear variables that appear. In order to ensure that linear variables are used, we move these checks to the rules where variables are introduced. For instance, consider the rule (A-SPLIT).

Algorithmic Typing $\boxed{\Gamma_{in} \vdash \texttt{t : T;}\Gamma_{out}}$

$$\Gamma_1, \texttt{x:un P}, \Gamma_2 \vdash \texttt{x : un P;}\Gamma_1, \texttt{x:un P}, \Gamma_2 \qquad \text{(A-UVAR)}$$

$$\Gamma_1, \texttt{x:lin P}, \Gamma_2 \vdash \texttt{x : lin P;}\Gamma_1, \Gamma_2 \qquad \text{(A-LVAR)}$$

$$\Gamma \vdash \texttt{q b : q Bool;}\Gamma \qquad \text{(A-BOOL)}$$

$$\frac{\Gamma_1 \vdash \texttt{t}_1 \texttt{ : q Bool;}\Gamma_2 \qquad \Gamma_2 \vdash \texttt{t}_2 \texttt{ : T;}\Gamma_3 \qquad \Gamma_2 \vdash \texttt{t}_3 \texttt{ : T;}\Gamma_3}{\Gamma_1 \vdash \texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \texttt{ : T;}\Gamma_3} \qquad \text{(A-IF)}$$

$$\frac{\Gamma_1 \vdash \texttt{t}_1 \texttt{ : T}_1\texttt{;}\Gamma_2 \qquad \Gamma_2 \vdash \texttt{t}_2 \texttt{ : T}_2\texttt{;}\Gamma_3 \qquad \texttt{q(T}_1\texttt{)} \qquad \texttt{q(T}_2\texttt{)}}{\Gamma_1 \vdash \texttt{q <t}_1\texttt{,t}_2\texttt{> : q (T}_1\texttt{*T}_2\texttt{);}\Gamma_3} \qquad \text{(A-PAIR)}$$

$$\frac{\Gamma_1 \vdash \texttt{t}_1 \texttt{ : q (T}_1\texttt{*T}_2\texttt{);}\Gamma_2 \qquad \Gamma_2, \texttt{x:T}_1, \texttt{y:T}_2 \vdash \texttt{t}_2 \texttt{ : T;}\Gamma_3}{\Gamma_1 \vdash \texttt{split t}_1 \texttt{ as x,y in t}_2 \texttt{ : T;}\Gamma_3 \div \texttt{(x:T}_1, \texttt{y:T}_2\texttt{)}} \qquad \text{(A-SPLIT)}$$

$$\frac{\texttt{q=un} \Rightarrow \Gamma_1 = \Gamma_2 \div \texttt{(x:T}_1\texttt{)} \qquad \Gamma_1, \texttt{x:T}_1 \vdash \texttt{t}_2 \texttt{ : T}_2\texttt{;}\Gamma_2}{\Gamma_1 \vdash \texttt{q } \lambda\texttt{x:T}_1\texttt{.t}_2 \texttt{ : q T}_1 \texttt{→} \texttt{T}_2\texttt{;}\Gamma_2 \div \texttt{(x:T}_1\texttt{)}} \qquad \text{(A-ABS)}$$

$$\frac{\Gamma_1 \vdash \texttt{t}_1 \texttt{ : q T}_{11} \texttt{→} \texttt{T}_{12}\texttt{;}\Gamma_2 \qquad \Gamma_2 \vdash \texttt{t}_2 \texttt{ : T}_{11}\texttt{;}\Gamma_3}{\Gamma_1 \vdash \texttt{t}_1 \texttt{ t}_2 \texttt{ : T}_{12}\texttt{;}\Gamma_3} \qquad \text{(A-APP)}$$

Figure 2-6: Linear lambda calculus: Algorithmic Type Checking

The second premise has the form

$$\Gamma_2, \texttt{x:T}_1, \texttt{y:T}_2 \vdash \texttt{t}_2 \texttt{ : T;}\Gamma_3$$

If $\texttt{T}_1$ and $\texttt{T}_2$ are linear, then they should be used in $\texttt{t}_2$, and should not appear in Γ_3. Conversely, $\texttt{T}_1$ and $\texttt{T}_2$ are unrestricted, then they will always appear in Γ_3, but we should delete them from the final outgoing context of the rule so that the ordinary scoping rules for the variables are enforced. To handle both the check that linear variables do not appear and the removal of unrestricted variables, we use a special "context difference" operator ($\div$). Using this operator, the final outgoing context of the rule (A-SPLIT) is defined to be $\Gamma_3 \div \texttt{(x:T}_1, \texttt{y:T}_2\texttt{)}$. Formally, context difference is defined as follows.

$$\Gamma \div \emptyset = \Gamma$$

$$\frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \qquad \texttt{(x:lin P)} \notin \Gamma_3}{\Gamma_1 \div (\Gamma_2, \texttt{x:lin P}) = \Gamma_3}$$

$$\frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \qquad \Gamma_3 = \Gamma_4, \texttt{x:un P}, \Gamma_5}{\Gamma_1 \div (\Gamma_2, \texttt{x:un P}) = \Gamma_4, \Gamma_5}$$

Notice that this operator is undefined when we attempt to take the difference of two contexts, Γ_1 and Γ_2, that contain bindings for the same linear variable ($\texttt{x:lin P}$). If the undefined quotient $\Gamma_1 \div \Gamma_2$ were to appear anywhere in a typing rule, the rule itself would not be considered defined and could not be part of a valid typing derivation.

The rule for abstraction (A-ABS) also introduces a variable and hence it also uses context difference to manipulate the output context for the rule. Abstractions must also satisfy the appropriate containment conditions. In other words, rule (A-ABS) must check that unrestricted functions do not contain linear variables. We perform this last check by verifying that when the function qualifier is unrestricted, the input and output contexts from checking the function body are the same. This equivalence check is sufficient because if a linear variable was used in the body of an unrestricted function (and hence captured in the function closure), that linear variable would not show up in the outgoing context.

It is completely straightforward to check that every rule in our algorithmic system is syntax directed and that all our auxiliary functions including context membership tests and context difference are easily computable. Hence, we need only show that our algorithmic system is equivalent to the simpler and more elegant declarative system specified in the previous section. The proof of equivalence can be a broken down into the two standard components: *soundness* and *completeness* of the algorithmic system with respect to the declarative system. However, before we can get to the main results, we will need to show that our algorithmic system satisfies some basic structural properties of its own. In the following lemmas, we use the notation $\mathcal{L}(\Gamma)$ and $\mathcal{U}(\Gamma)$ to refer to the list of linear and unrestricted assumptions in Γ respectively.

2.2.5 LEMMA [ALGORITHMIC MONOTONICITY]: If $\Gamma \vdash \mathtt{t} : \mathtt{T};\Gamma'$ then $\mathcal{U}(\Gamma') = \mathcal{U}(\Gamma)$ and $\mathcal{L}(\Gamma') \subseteq \mathcal{L}(\Gamma)$. □

2.2.6 LEMMA [ALGORITHMIC EXCHANGE]: If $\Gamma_1, \mathtt{x}_1\mathtt{:T}_1, \mathtt{x}_2\mathtt{:T}_2, \Gamma_2 \vdash \mathtt{t} : \mathtt{T};\Gamma_3$ then $\Gamma_1, \mathtt{x}_2\mathtt{:T}_2, \mathtt{x}_1\mathtt{:T}_1, \Gamma_2 \vdash \mathtt{t} : \mathtt{T};\Gamma_3'$ and Γ_3 is the same as Γ_3' up to transposition of the bindings for x_1 and x_2. □

2.2.7 LEMMA [ALGORITHMIC WEAKENING]: If $\Gamma \vdash \mathtt{t} : \mathtt{T};\Gamma'$ then $\Gamma, \mathtt{x:T}' \vdash \mathtt{t} : \mathtt{T};\Gamma', \mathtt{x:T}'$. □

2.2.8 LEMMA [ALGORITHMIC LINEAR STRENGTHENING]: If $\Gamma, \mathtt{x:lin\,P} \vdash \mathtt{t} : \mathtt{T};\Gamma', \mathtt{x:lin\,P}$ then $\Gamma \vdash \mathtt{t} : \mathtt{T};\Gamma'$. □

Each of these lemmas may be proven directly by induction on the initial typing derivation. The algorithmic system also satisfies a contraction lemma, but since it will not be necessary in the proofs of soundness and completeness, we have not stated it here.

`w ::=`		*prevalues:*
	`b`	*boolean*
	`<x,y>`	*pair*
	`λx:T.t`	*abstraction*
`v ::=`		*values:*
	`q w`	*qualified prevalue*

`S ::=`		*stores:*
	$\emptyset$	*empty context*
	`S, x` $\mapsto$ `v`	*store binding*
`E ::=`		*evaluation contexts:*
	`[ ]`	*context hole*
	`if E then t else t`	*if context*
	`q <E,t>`	*fst context*
	`q <x,E>`	*snd context*
	`split E as x,y in t`	*split context*
	`E t`	*fun context*
	`x E`	*arg context*

Figure 2-7: Linear lambda calculus: Run-time Data

2.2.9 THEOREM [ALGORITHMIC SOUNDNESS]: If $\Gamma_1 \vdash$ `t : T`;Γ_2 and $\mathcal{L}(\Gamma_2) = \emptyset$ then $\Gamma_1 \vdash$ `t : T`. □

Proof: As usual, the proof is by induction on the typing derivation. The structural lemmas we have just proven are required to push through the result, but it is mostly straightforward. □

2.2.10 THEOREM [ALGORITHMIC COMPLETENESS]: If $\Gamma_1 \vdash$ `t : T` then $\Gamma_1 \vdash$ `t : T`;Γ_2 and $\mathcal{L}(\Gamma_2) = \emptyset$. □

Proof: The proof is by induction on the typing derivation. □

Operational Semantics

To make the memory management properties of our language clear, we will evaluate terms in an abstract machine with an explicit store. As indicated in Figure 2-7, stores are a sequence of variable-value pairs. We will implicitly assume that any variable appears at most once on the left-hand side of a pair so the sequence may be treated as a finite partial map.

A value is a pair of a qualifier together with some data (a *prevalue* `w`). For the sake of symmetry, we will also assume that all values are stored, even base types such as booleans. As a result, both components of any pair will be pointers (variables).

We define the operation of our abstract machine using a context-based, small-step semantics. Figure 2-8 defines the computational contexts `E`, which are terms with a single hole. Contexts define the order of evaluation of terms—they specify the places in a term where a computation can occur. In our case, evaluation is left-to-right since, for example, there is a context with the form `E t` indicating that we can reduce the term in the function position before reducing the term in the argument position. However, there is no context with the form `t E`. Instead, there is only the more limited context `x E`, indicating that we must reduce the term in the function position to a pointer `x` before proceeding to evaluate the term in the argument position. We use the notation `E[t]` to denote the term composed of the context `E` with its hole plugged by the computation `t`.

The operational semantics, defined in Figure ♠??, is factored into two relations. The first relation, $(S;t) \longrightarrow (S';t')$, picks out a subcomputation to evaluate. The second relation, $(S;t) \longrightarrow_\beta (S';t')$, does all the real work. In order to avoid creation of two sets of operational rules, one for linear data, which is deallocated when used, and one for unrestricted data, which is never deallocated, we define an auxiliary function, $S \overset{q}{\sim} x$, to manage the differences.

$$
\begin{array}{lcl}
(S_1, x \mapsto v, S_2) \overset{\texttt{lin}}{\sim} x & = & S_1, S_2 \\
S \overset{\texttt{un}}{\sim} x & = & S
\end{array}
$$

Aside from these details, the operational semantics is standard.

Preservation and Progress

In order to prove the standard safety properties for our language, we need to be able to show that programs are well-formed after each step in evaluation. Hence, we will define typing rules for our abstract machine. Since these typing rules are only necessary for the proof of soundness, and have no place in an implementation, we will extend the declarative typing rules rather than the algorithmic typing rules.

Figure 2-9 presents the machine typing rules in terms of two judgments, one for stores and the other for programs. The store typing rules generate a context that describes the available bindings in the store. The program typing rule uses the generated bindings to check the expression that will be executed.

With this new machinery in hand, we are able to prove the standard progress and preservation theorems.

2.2.11 THEOREM [PRESERVATION]: If $\vdash (S;t)$ and $(S;t) \longrightarrow (S';t')$ then $\vdash (S';t')$. □

Top-level Evaluation $\boxed{(S;t) \longrightarrow (S';t')}$

$$\frac{(S;t) \longrightarrow_\beta (S;t')}{(S;E[t]) \longrightarrow (S;E[t'])} \quad \text{(E-CTXT)}$$

Evaluation $\boxed{(S;t) \longrightarrow_\beta (S';t')}$

$$(S;q\ b) \longrightarrow_\beta (S, x \mapsto q\ b; x) \quad \text{(E-BOOL)}$$

$$\frac{S(x) = q\ \mathtt{true}}{(S;\mathtt{if}\ x\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2) \longrightarrow_\beta (S \overset{q}{\sim} x; t_1)} \quad \text{(E-IF1)}$$

$$\frac{S(x) = q\ \mathtt{false}}{(S;\mathtt{if}\ x\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2) \longrightarrow_\beta (S \overset{q}{\sim} x; t_2)} \quad \text{(E-IF2)}$$

$$(S;q <t_1,t_2>) \longrightarrow_\beta (S, x \mapsto q <t_1,t_2>; x) \quad \text{(E-PAIR)}$$

$$\frac{S(x) = q <y_1,z_1>}{(S;\mathtt{split}\ x\ \mathtt{as}\ y,z\ \mathtt{in}\ t) \longrightarrow_\beta (S \overset{q}{\sim} x; [y \mapsto y_1][z \mapsto z_1]t)} \quad \text{(E-SPLIT)}$$

$$(S;q\ \lambda y{:}T.t) \longrightarrow_\beta (S, x \mapsto q\ \lambda y{:}T.t; x) \quad \text{(E-FUN)}$$

$$\frac{S(x_1) = q\ \lambda y{:}T.t}{(S;x_1\ x_2) \longrightarrow_\beta (S \overset{q}{\sim} x_1; [y \mapsto x_2]t)} \quad \text{(E-APP)}$$

Figure 2-8: Linear lambda calculus: Operational semantics

Store Typing $\boxed{\vdash S : \Gamma}$

$$\vdash \emptyset : \emptyset \quad \text{(T-EMPTYS)}$$

$$\frac{\vdash S : \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{lin}\ w : T}{\vdash S, x \mapsto \mathtt{lin}\ w : \Gamma_2, x{:}T} \quad \text{(T-NEXTLINS)}$$

$$\frac{\vdash S : \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{un}\ w : T}{\vdash S, x \mapsto \mathtt{un}\ w : \Gamma_2, x{:}T} \quad \text{(T-NEXTUNS)}$$

Program Typing $\boxed{\vdash (S;t)}$

$$\frac{\vdash S : \Gamma \qquad \Gamma \vdash t : T}{\vdash (S;t)} \quad \text{(T-PROG)}$$

Figure 2-9: Linear lambda calculus: Program Typing

2.2.12 THEOREM [PROGRESS]: If $\vdash (S;t)$ then $(S;t) \longrightarrow (S';t')$ or t is a value. □

2.2.13 EXERCISE [★★, ↛]: Prove progress and preservation using TAPL Chapters 9,13 as an approximate guide. □

2.3 Extensions and Variations

Most features found in modern programming languages can be defined to interoperate successfully with linear type systems, although some are trickier than others. In this section, we will consider a variety of practical extensions to our simple linear lambda calculus.

Sums and Recursive Types

Complex data structures, such as the recursive data types found in ML-like languages, pose little problem for linear languages. To demonstrate the central ideas involved, we extend the syntax for the linear lambda calculus with the standard introduction and elimination forms for sums and recursive types. The details are presented in Figure 2-10.

Values with sum type are introduced by injections `q inl`$_P$ `t` or `q inr`$_P$ `t`, where `P` is T_1+T_2, the resulting pretype of the term. In the first instance, the underlying term `t` must have type T_1, and in the second instance, the underlying term `t` must have type T_2. The qualifier `q` indicates the linearity of the argument in exactly the same way as for pairs. The case expression will execute its first branch if its primary argument is a left injection and it will execute its second branch if its primary argument is a right injection. We will assume that `+` binds more tightly that $\rightarrow$ but less tightly than `*`.

Recursive types are introduced with a `roll`$_P$ `t` expression, where `P` is the recursive pretype the expression will assume. Unlike all the other introduction forms, roll expressions are not annotated with a qualifier. Instead, they take on the qualifier of the underlying expression `t`. The reason for this distinction is that we will treat this introduction form as a typing coercion that has no real operational effect. Unlike functions, pairs or sums, recursive data types have no data of their own and therefore do not need a separate qualifier to control their allocation behavior. To simplify the notational overhead of sums and recursive types, we will normally omit the typing annotations on their introduction forms in our examples.

In order to write computations that process recursive types, we have added recursive function declarations to our language as well. Since the free variables in a recursive function closure will be used on each recursive invocation of the function, we cannot allow the closure to contain linear variables. Hence, all recursive functions are unrestricted data structures.

A simple, but useful data structure is the linear list of `T`s:

```
type T llist = rec a.lin (unit + lin (T * lin a))
```

Here, the entire spine (aside from the terminating value of unit type) is linear while the underlying `T` objects may be linear or unrestricted. To create a fully unrestricted list, we simply omit the linear qualifiers on the sum and pairs that make up the spine of the list:

```
type T list = rec a.unit + T * a
```

`t ::=`		*terms:*
	`...`	*as before*
	$\mathtt{q\ inl_P\ t}$	*left inj.*
	$\mathtt{q\ inr_P\ t}$	*right inj.*
	$\mathtt{case\ t\ (inl\ x \Rightarrow t \mid inr\ y \Rightarrow t)}$	*case*
	$\mathtt{roll_P\ t}$	*roll into rec type*
	`unroll t`	*unroll from rec type*
	$\mathtt{fun\ f(x{:}T_1){:}T_2.t}$	*recursive fun*
`P ::=`		*pretypes:*
	`...`	*as before*
	`a`	*pretype variables*
	$\mathtt{T_1{+}T_2}$	*sum types*
	`rec a.T`	*recursive types*

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T_1} \qquad \mathtt{q(T_1)} \qquad \mathtt{q(T_2)}}{\Gamma \vdash \mathtt{q\ inl_{T_1+T_2}\ t} : \mathtt{q\ (T_1{+}T_2)}} \qquad \text{(T-INL)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T_2} \qquad \mathtt{q(T_1)} \qquad \mathtt{q(T_2)}}{\Gamma \vdash \mathtt{q\ inr_{T_1+T_2}\ t} : \mathtt{q\ (T_1{+}T_2)}} \qquad \text{(T-INR)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \mathtt{t} : \mathtt{q\ (T_1{+}T_2)} \\ \Gamma_2, \mathtt{x{:}T_1} \vdash \mathtt{t_1} : \mathtt{T} \qquad \Gamma_2, \mathtt{y{:}T_2} \vdash \mathtt{t_2} : \mathtt{T}\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{case\ t\ (inl\ x \Rightarrow t_1 \mid inr\ y \Rightarrow t_2)} : \mathtt{T}} \qquad \text{(T-CASE)}$$

$$\frac{\Gamma \vdash \mathtt{t} : [\mathtt{a} \mapsto \mathtt{P}]\mathtt{q\ P_1} \qquad \mathtt{P = rec\ a.q\ P_1}}{\Gamma \vdash \mathtt{roll_P\ t} : \mathtt{q\ P}} \qquad \text{(T-ROLL)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{P} \qquad \mathtt{P = rec\ a.q\ P_1}}{\Gamma \vdash \mathtt{unroll\ t} : [\mathtt{a} \mapsto \mathtt{P}]\mathtt{q\ P_1}} \qquad \text{(T-UNROLL)}$$

$$\frac{\mathtt{un}\ (\Gamma) \qquad \Gamma, \mathtt{f{:}un\ T_1{\to}T_2}, \mathtt{x{:}T_1} \vdash \mathtt{t} : \mathtt{T_2}}{\Gamma \vdash \mathtt{fun\ f(x{:}T_1){:}T_2.t} : \mathtt{un\ T_1{\to}T_2}} \qquad \text{(T-TFUN)}$$

Figure 2-10: Linear lambda calculus: Sums and Recursive Types

2.3.1 EXERCISE [★]: What is "wrong" with the following list type declaration. Does this example reveal an error in our language design?

```
type T badlist = rec a.lin (unit + T * a)
```

□

Solution?

After defining the linear lists, the memory conscious programmer can write many familiar list-processing functions in a minimal amount of space. For example, here is how we map an unrestricted function across a linear list.

```
fun nil(_:unit) : T2 llist =
   roll (lin inl ())

fun cons(hd:T2, tl:T2 llist) : T2 llist =
   roll (lin inr (lin <hd,tl>))
```

```
fun map(f:T₁→T₂, xs:T₁ llist) : T₂ llist =
  case unroll xs (
    inl _  ⇒ nil()
  | inr xs ⇒
      split xs as hd,tl in
      cons(f hd,map lin <f,tl>))
```

In this implementation of `map`, we can observe that on each iteration of the loop, it is possible to reuse the space deallocated by `split` or `case` operations for the allocation operations that follow in the body of the function (inside the calls to `nil` and `cons`).

Hence, at first glance, it appears that `map` will execute with only a constant space overhead. Unfortunately, however, there are some hidden costs as `map` executes. A typical implementation will store local variables and temporaries on the stack before making a recursive call. In this case, the result of `f hd` will be stored on the stack while map iterates down the list. Consequently, rather than having a constant space overhead, our map implementation will have an $O(n)$ overhead, where n is the length of the list. This is not too bad, but we can do better.

In order to do better, we need to avoid implicit stack allocation of data each time we iterate through the body of a recursive function. Fortunately, most functional programming languages guarantee that if the last operation in a function is itself a function call then the language implementation will deallocate the current stack frame before calling the new function. We name such function calls *tail calls* and we say that any language implementation that guarantees that the current stack frame will be deallocated before a tail call is *tail-call optimizing*.

Assuming that our language is tail-call optimizing, we can now rewrite map so that it executes with only a constant space overhead. The main trick involved is that we will explicitly keep track of both the part of the input list we have yet to process and the ouput list that we have already processed. The output list will wind up in reverse order, so we will reverse it at the end. Both of the loops in the code, `mapRev` and `reverse` are *tail-recursive* functions. That is, they end in a tail call and have a space-efficient implementation.

```
fun map(f:T₁→T₂, input:T₁ llist) : T₂ llist =
  reverse(mapRev(f,input,nil()),nil())

and mapRev(f:T₁→T₂,
           input:T₁ llist,
           output:T₂ llist) : T₂ llist =
  case unroll input (
      inl _  ⇒ output
    | inr xs ⇒
        split xs as hd,tl in
        mapRev (f,tl,cons(f hd,output)))

and reverse(input:T₂ llist, output:T₂ llist)
    case unroll input (
      inl _  ⇒ output
    | inr xs ⇒
        split xs as hd,tl in
        reverse(tl,cons(hd,output)))
```

This *link reversal* algorithm is a well-known way of traversing a list in constant space. It is just one of a class of algorithms developed well before the invention of linear types. A similar algorithm was invented by Deutsch, Schorr and Waite for traversing trees and graphs in constant space. Such constant space traversals are essential parts of mark-sweep garbage collectors—at garbage collection time there is no extra space for a stack so any traversal of the heap must be done in constant space.

2.3.2 EXERCISE [★★★]: Define a recursive type that describes linear binary trees that hold data of type `T` in their internal nodes (nothing at the leaves). Write a constant-space function `treeMap` that produces an identically-shaped tree on output as it was given on input, modulo the action of the function `f` that is applied to each element of the tree. Feel free to use reasonable extensions to our linear lambda calculus including mutually recursive functions, n-ary tuples and n-ary sums. □

Polymorphism

Parametric polymorphism is a crucial feature of almost any functional language, and our linear lambda calculus is no exception. The main function of polymorphism in our setting is to support two different sorts of code reuse.

1. Reuse of code to perform the same algorithm, but on data with different shapes.

2. Reuse of code to perform the same algorithm, but on data governed by different memory management strategies.

To support the first kind of polymorphism, we will allow quantification over pretypes. To support the second kind of polymorphism, we will allow quantification over qualifiers. A good example of both sorts of polymorphism arises in the definition of a polymorphic `map` function. In the code

`q` ::=		*qualifiers:*
	`...`	*as before*
	`p`	*polymorphic qualifier*
`t` ::=		*terms:*
	`...`	*as before*
	`q Λa.t`	*pretype abstraction*
	`t [P]`	*pretype application*
	`q Λp.t`	*qualifier abstraction*
	`t [q]`	*qualifier application*
`P` ::=		*pretypes:*
	`...`	*as before*
	`∀a.T`	*pretype polymorphism*
	`∀p.T`	*qualifier polymorphism*

Figure 2-11: Linear lambda calculus: Polymorphism Syntax

below, we use `a` and `b` to range over pretype variables as we did in the previous section, and `p` to range over qualifier variables.

```
type (p1,p2,a) list =
  rec a.p1 (unit + p1 (p2 a * (p1,p2,a) list))

map :
  ∀a,b.
    ∀pa,pb.
      lin ((pa a → pb b)*(lin,pa,a) list)→(lin,pb,b) list
```

The type definition in the first line defines lists in terms of three parameters. The first parameter, p_1, gives the usage pattern (linear or unrestricted) for the spine of the list, while the second parameter gives the usage pattern for the elements of the list. The third parameter is a pretype parameter, which gives the (pre)type of the elements of list. The `map` function is polymorphic in the argument (`a`) and result (`b`) element types of the list. It is also polymorphic (via parameters p_a and p_b) in the way those elements are used. Overall, the function maps lists with linear spines to lists with linear spines.

Developing a system for polymorphic, linear type inference is a challenging research topic, beyond the scope of this book, so we will assume that, unlike in ML, polymorphic functions are introduced explicitly using the syntax `Λa.t` or `Λp.t`. Here, `a` or `p` are the type parameters to a function with body `t`. The body does not need to be a value, like in ML, since we will run the polymorphic function every time a pretype or qualifier is passed to the function as an argument. The syntax `t′ [P]` or `t′ [q]` applies the function `t′` to its pretype or qualifier argument. Figure 2-11 summarizes the syntactic extensions to the language.

Before we get to writing the `map` function, we will take a look at the polymorphic constructor functions for linear lists. These functions will take a pretype parameter and two qualifier parameters, just like the type definition for lists.

```
val nil : ∀a,p₂.(lin,p₂,a) list =
  Λa,p₂.roll (lin inl ())

val list :
  ∀a,p₂.lin (p₂ a * (lin,p₂,a) list)→(lin,p₂,a) list =
  Λa,p₂.
    λcell : lin (p₂ a * (lin,p₂,a) list).
      roll (lin inr (lin cell))
```

Now our most polymorphic `map` function may be written as follows.

```
val map =
  Λa,b.
  Λpₐ,p_b.
  fun aux(f:(pₐ a → p_b b),
          xs:(lin,pₐ,a) list)) : (lin,p_b,b) list =
    case unroll xs (
      inl _  ⇒ nil [b,p_b] ()
    | inr xs ⇒
        split xs as hd,tl in
         cons [b,p_b] (p_b <f hd,map (lin <f,tl>)>))
```

In order to ensure that our type system remains sound in the presence of pretype polymorphism, we will add the obvious typing rules, but change very little else. However, adding qualifier polymorphism, as we have done, is a little more involved. Before arriving at the typing rules themselves, we need to adapt some of our basic definitions to account for abstract qualifiers that may either be linear or unrestricted.

First, we need to ensure that we propagate contexts containing abstract qualifiers safely through the other typing rules in the system. Most importantly, we will add additional cases to the context manipulation rules defined in the previous section. We need to ensure that linear hypotheses are not duplicated and therefore we cannot risk duplicating unknown qualifiers, which might turn out to be linear. Figure 2-12 specifies the details.

Second, we need to conservatively extend the relation on type qualifiers $q_1 \sqsubseteq q_2$ so that it is sound in the presence of qualifier polymorphism. Since the linear qualifier is the least qualifier in the current system, the following rule should hold.

$$\texttt{lin} \sqsubseteq \texttt{p} \qquad \text{(Q-LinP)}$$

Likewise, since `un` is the greatest qualifier in the system, we can be sure the following rule is sound.

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}p\ P} = (\Gamma_1, \mathtt{x{:}p\ P}) \circ \Gamma_2} \quad \text{(M-ABS1)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}p\ P} = \Gamma_1 \circ (\Gamma_2, \mathtt{x{:}p\ P})} \quad \text{(M-ABS2)}$$

Figure 2-12: Linear Context Manipulation Rules

Δ ::= *type contexts:*
- $\emptyset$ *empty*
- $\Delta, \mathtt{a}$ *pretype var.*
- $\Delta, \mathtt{p}$ *qualifier var.*

Typing $\boxed{\Delta; \Gamma \vdash \mathtt{t : T}}$

$$\frac{\mathtt{q}(\Gamma) \qquad \Delta, \mathtt{a}; \Gamma \vdash \mathtt{t : T}}{\Delta; \Gamma \vdash \mathtt{q}\ \Lambda\mathtt{a.t : q}\ \forall\mathtt{a.T}} \quad \text{(T-PABS)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{t : q}\ \forall\mathtt{a.T} \qquad FV(\mathtt{P}) \subseteq \Delta}{\Delta; \Gamma \vdash \mathtt{t\ [P] : [a \mapsto P]T}} \quad \text{(T-PAPP)}$$

$$\frac{\mathtt{q}(\Gamma) \qquad \Delta, \mathtt{p}; \Gamma \vdash \mathtt{t : T}}{\Delta; \Gamma \vdash \mathtt{q}\ \Lambda\mathtt{p.t : q}\ \forall\mathtt{p.T}} \quad \text{(T-QABS)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{t : q_1}\ \forall\mathtt{p.T} \qquad FV(\mathtt{q}) \subseteq \Delta}{\Delta; \Gamma \vdash \mathtt{t\ [q] : [p \mapsto q]T}} \quad \text{(T-QAPP)}$$

Figure 2-13: Linear lambda calculus: Polymorphic Typing

$$\mathtt{p} \sqsubseteq \mathtt{un} \qquad \text{(Q-PUN)}$$

Aside from these rules, we will only be able to infer that an abstract qualifier p is related to itself via the general reflexivity rule. Consequently, linear data structures can contain abstract ones; abstract data structures can contain unrestricted data structures; and data structure with qualifier p can contain other data with qualifier p.

In order to define the typing rules for the polymorphic linear lambda calculus proper, we will need to change the judgment form to keep track of the type variables that are allowed to appear free in a term. The new judgment uses the type context Δ for this purpose. The typing rules for the introduction and elimination forms for each sort of polymorphism are fairly straightforward now and are presented in Figure 2-13.

The typing rules for the other constructs we have seen will be almost unchanged. One relatively minor alteration is that the incoming type context Δ will be propagated through the rules to account for the free type variables. Unlike term variables, type variables can always be used in an unrestricted fashion; it is difficult to understand what it would mean to restrict the use of a type variable to one place in another type or term. Consequently, all parts of Δ are propagated from the conclusion of any rule to all premises. We will also need the occasional side condition to check that whenever a program-

$$\begin{array}{lr} \texttt{E} ::= & \textit{evaluation contexts:} \\ \quad \texttt{E [P]} & \textit{pretype app context} \\ \quad \texttt{E [q]} & \textit{qualifier app context} \end{array}$$

$$(\texttt{S;q}\,\Lambda\texttt{a.t}) \longrightarrow_\beta (\texttt{S}, \texttt{x} \mapsto \texttt{q}\,\Lambda\texttt{a.t;x}) \qquad \text{(E-PFun)}$$

$$\frac{\texttt{S(x)} = \texttt{q}\,\Lambda\texttt{a.t}}{(\texttt{S;x [P]}) \longrightarrow_\beta (\texttt{S} \overset{\texttt{q}}{\sim} \texttt{x;}[\texttt{a} \mapsto \texttt{P}]\texttt{t})} \qquad \text{(E-PApp)}$$

$$(\texttt{S;q}\,\Lambda\texttt{p.t}) \longrightarrow_\beta (\texttt{S}, \texttt{x} \mapsto \texttt{q}\,\Lambda\texttt{p.t;x}) \qquad \text{(E-QFun)}$$

$$\frac{\texttt{S(x)} = \texttt{q}\,\Lambda\texttt{p.t}}{(\texttt{S;x [q}_1\texttt{]}) \longrightarrow_\beta (\texttt{S} \overset{\texttt{q}}{\sim} \texttt{x;}[\texttt{p} \mapsto \texttt{q}_1]\texttt{t})} \qquad \text{(E-QApp)}$$

Figure 2-14: Linear lambda calculus: Polymorphic Operational Semantics

mer writes down a type, its free variables are contained in the current type context Δ. For instance the rules for function abstraction and application will now be written as follows.

$$\frac{\texttt{q}(\Gamma) \qquad FV(\texttt{T}_1) \subseteq \Delta \qquad \Delta;\Gamma,\texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Delta;\Gamma \vdash \texttt{q}\,\lambda\texttt{x:T}_1\texttt{.t}_2 : \texttt{q T}_1 \rightarrow \texttt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Delta;\Gamma_1 \vdash \texttt{t}_1 : \texttt{q T}_1 \rightarrow \texttt{T}_2 \qquad \Delta;\Gamma_2 \vdash \texttt{t}_2 : \texttt{T}_1}{\Delta;\Gamma_1 \circ \Gamma_2 \vdash \texttt{t}_1\ \texttt{t}_2 : \texttt{T}_2} \qquad \text{(T-App)}$$

The most important check we need to do to test our system for faults is to prove the type substitution lemma. In particular, the proof will demonstrate that we have made safe assumptions about how abstract type qualifiers may be used.

2.3.3 Lemma [Type Substitution]: 1. If Δ, $p;\Gamma \vdash \texttt{t} : \texttt{T}$ and $FV(\texttt{q}) \in \Delta$ then $\Delta;[\texttt{p} \mapsto \texttt{q}]\Gamma \vdash [\texttt{p} \mapsto \texttt{q}]\texttt{t} : [\texttt{p} \mapsto \texttt{q}]\texttt{T}$

2. If Δ, $\texttt{a};\Gamma \vdash \texttt{t} : \texttt{T}$ and $FV(\texttt{P}) \in \Delta$ then $\Delta;[\texttt{a} \mapsto \texttt{P}]\Gamma \vdash [\texttt{a} \mapsto \texttt{P}]\texttt{t} : [\texttt{a} \mapsto \texttt{P}]\texttt{T}$ □

2.3.4 Exercise [★]: Sketch the proof of the type substitution lemma. What structural rule(s) do you need to carry out the proof? □

Operationally, we will choose to implement polymorphic instantiation using substitution. As a result, our operational semantics changes very little. We only need to specify the new computational contexts and to add the evaluation rules for polymorphic functions and application as in Figure 2-14.

Arrays

Arrays pose a special problem for linearly typed languages. If we try to provide an operation fetches an element from an array in the usual way, perhaps using an array index expression `a[i]`, we would need to reflect the fact that the `i`$^{\texttt{th}}$ element (and only the `i`$^{\texttt{th}}$ element) of the array had been "used." However, there is no simple way to reflect this change in the type of an array as the usual form of array types (`array(T)`) provides no mechanism to distinguish between the properties of different elements of the array.

We dodged this problem when we constructed our tuple operations by defining a pattern matching construct that simultaneously extracted all of the elements of a tuple. Unfortunately, we cannot follow the same path for arrays because in modern languages like Java and ML, the length of an array (and therefore the size of the pattern) is unknown at compile time.

Yet another non-solution to the problem is to add a special built-in iterator to process all the elements in an array at once. However, this last straw man proposal prevents programmers from using arrays as efficient, constant-time, random-access data structures; they might as well use lists instead.

One way out of this jam is to design the central array access operations so that, unlike the ordinary "get" and "set" operations, they *preserve* the number of pointers to the array and the number of pointers to each of its elements. We avoid our problem because there is no change to the array data structure that needs to be reflected in the type system. Using this idea, we will be able to allow programmers to define linear arrays that can hold a collection of arbitrarily many linear objects. Moreover, programmers will be able to access any of these linear objects, one at a time, using a convenient, constant-time, random-access mechanism.

So, what are the magic pointer-preserving array access operations? Actually, we need only one: a swap operation with the form `swap(a[i],t)`. The swap replaces the `i`$^{\texttt{th}}$ element of the array `a` (call it `t'`) with `t` and returns a (linear) pair containing the new array and `t'`. Notice the number of pointers to `t` and `t'` does not change during the operation. If there was one pointer to `t` (as an argument to `swap`) before the call, then there is one pointer to `t` afterward (from within the array `a`) and vice versa for `t'`. If, in addition, all of the elements of `a` had one pointer to them before the swap, then they will all have one pointer to them after the swap as well. Consequently, we will find it easy to type the swap operation, even when it works over linear arrays of linear objects.

In addition to swap, we provide functions to allocate an array given its list of elements (`array`), to determine array length (`length`) and to deallocate arrays (`free`). The last operation is somewhat unusual in that it takes two arguments a and f, where a is an array of type lin array(T) and f is a function with type T→unit that is run on each element of T. The function may be thought of as a finalizer for the elements; it may be used to deallocate any linear components of the array elements, thereby preserving the single pointer property.

Our definition of arrays is compatible with the polymorphic system from the previous subsection, but for simplicity, we present the formal syntax and semantics in the context of the simply-typed lambda calculus (see Figure 2-15).

2.3.5 EXERCISE [★,RECOMMENDED]: The typing rule for array allocation (T-ARRAY) contains the standard containment check to ensure that unrestricted arrays cannot contain linear objects. What kinds of errors can occur if this check is omitted? □

The swap and free functions are relatively low-level operations. Fortunately, it is easy to build more convenient, higher-level abstractions out of them. For instance, the following code defines some simple functions for manipulating linear matricies of unrestricted integers.

```
type iArray = lin array(int)
type matrix = lin array(iArray)

fun dummy(x:unit):iArray = lin array()

fun freeElem(x:int):unit = ()
fun freeArray(a:iArray):unit = free(a,freeElem)
fun freeMatrix(m:matrix):unit = free(m,freeArray)

fun get(a:matrix,i:int,j:int):lin (matrix * int) =
  split swap(a[i],dummy()) as a,b in
  split swap(b[j],0) as b,k in
  split swap(b[j],k) as b,_ in
  split swap(a[i],b) as a,junk in
  freeArray(junk);
  lin <a,k>

fun set(a:matrix,i:int,j:int,e:int):matrix =
  split swap(a[i],dummy()) as a,b in
  split swap(b[j],e) as b,_ in
  split swap(a[i],b) as a,junk in
  freeArray(junk);
  a
```

`P`	::=		*pretypes:*
		`...`	*as before*
		`array(T)`	*array pretypes*
`t`	::=		*terms:*
		`...`	*as before*
		`q array(t,...,t)`	*array creation*
		`swap(t[t],t)`	*swap*
		`length(t)`	*length*
		`free(t,t)`	*deallocate*
`w`	::=		*prevalues:*
		`...`	*as before*
		`array[n,x,...,x]`	*array*
`E`	::=		*evaluation contexts:*
		`...`	*as before*
		`q array(v,...,v,E,t,...,t)`	*array context*
		`swap(E(t),t)`	*swap context*
		`swap(v(E),t)`	*swap context*
		`swap(v(v),E)`	*swap context*
		`length(E)`	*length context*
		`free(E,t)`	*free context*
		`free(v,E)`	*free context*

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{q(T)} \qquad \Gamma \vdash \mathtt{t}_i : \mathtt{T} \qquad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash \mathtt{q\ array(t_1,\ldots,t_n)} : \mathtt{q\ array(T)}} \quad \text{(T-ARRAY)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t_1} : \mathtt{q_1\ array(T_1)} \\ \Gamma \vdash \mathtt{t_2} : \mathtt{q_2\ int} \qquad \Gamma \vdash \mathtt{t_3} : \mathtt{T_1}\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{swap(t_1[t_2],t_3)} : \\ \mathtt{lin\ (q_1\ array(T_1)\ *\ T_1)}\end{array}} \quad \text{(T-SWAP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{q\ array(T)}}{\Gamma \vdash \mathtt{length(t)} : \mathtt{lin\ (q\ array(T)\ *\ int)}} \quad \text{(T-LENGTH)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{q\ array(T)} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T} \to \mathtt{unit}}{\Gamma \vdash \mathtt{free(t_1,t_2)} : \mathtt{unit}} \quad \text{(T-FREE)}$$

Evaluation $\boxed{(\mathtt{S;t}) \longrightarrow_\beta (\mathtt{S';t'})}$

$$\begin{array}{l}(\mathtt{S;q\ array(x_0,\ldots,x_{n-1})}) \longrightarrow_\beta \\ \quad ((\mathtt{S}, \mathtt{x} \mapsto \mathtt{q\ array[n,x_0,\ldots,x_{n-1}]};\mathtt{x})\end{array} \quad \text{(E-ARRAY)}$$

$$\frac{\begin{array}{c}\mathtt{S(x_i)} = \mathtt{q_i\ j} \\ \mathtt{S} = \mathtt{S_1}, \mathtt{x_a} \mapsto \mathtt{q\ array[n,\ldots,x_j,\ldots]}, \mathtt{S_2} \\ \mathtt{S'} = \mathtt{S_1}, \mathtt{x_a} \mapsto \mathtt{q\ array[n,\ldots,x_e,\ldots]}, \mathtt{S_2}\end{array}}{\begin{array}{l}(\mathtt{S;\ swap(x_a[x_i],x_e)}) \\ \quad \longrightarrow_\beta (\mathtt{S'} \overset{\mathtt{q_i}}{\sim} \mathtt{x_i};\mathtt{lin\ {<}x_a,x_j{>}})\end{array}} \quad \text{(E-SWAP)}$$

$$\frac{\mathtt{S(x)} = \mathtt{q\ array[n,x_0,\ldots,x_n]}}{(\mathtt{S;length(x)}) \longrightarrow_\beta (\mathtt{S;lin\ {<}x,un\ n{>}})} \quad \text{(E-LENGTH)}$$

$$\frac{\mathtt{S(x_a)} = \mathtt{q\ array[n,x_0,\ldots,x_n]}}{\begin{array}{l}(\mathtt{S;free(x_a,x_f)}) \\ \quad \longrightarrow_\beta (\mathtt{S} \overset{\mathtt{q}}{\sim} \mathtt{x_a};\mathtt{App(x_f,x_0,\ldots,x_{n-1})})\end{array}} \quad \text{(E-FREE)}$$

where

$$\begin{array}{lcl}\mathtt{App(x_f,\cdot)} & = & \mathtt{()} \\ \mathtt{App(x_f,x_1,\ldots)} & = & \mathtt{x_f\ x_1;App(x_f,\ldots)}\end{array}$$

Figure 2-15: Linear lambda calculus: Arrays

2.3.6 EXERCISE [★★, ↛]: Use the functions provided above to write matrix-matrix multiply. Your multiply function should return an integer and deallocate both arrays in the process. Use any standard integer operations necessary. □

In the examples above, we needed some sort of dummy value to swap into an array to replace the value we wanted to extract. For integers and arrays it was easy to come up with a dummy value. However, when dealing with polymorphic or abstract types, it may not be possible to conjure up a dummy value of the right type. Consequently, rather than manipulating arrays with type `q array(a)` for some abstract type `a`, we may need to manipulate arrays of options with type `q array(a + unit)`. In this case, when we need to read out a value, we always have another value (`inr ()`) to swap in in its place. Normally such operations are called *destructive reads*; they are quite a common way to preserve the single pointer property when managing complex structured data.

Reference Counting

Array swaps and destructive reads are dynamic techniques that can help overcome a lack of compile-time knowledge about the number of uses of a particular object. *Reference counting* is another dynamic technique that serves a similar purpose. Rather than restricting the number of pointers to an object to be exactly one, we can allow any number of pointers to the object and keep track of that number dynamically. Only when the last reference is used will the object be deallocated.

There are various ways to integrate reference counts into the current system. Here, we choose the simplest, which is to add a new qualifier `rc` for reference-counted data structures, and operations that allow the programmer to explicitly increment (`inc`) and decrement (`dec`) the counts. More specifically, the increment operation takes a pointer argument, increments the reference count for the object pointed to, and returns two copies of the pointer as a (linear) pair. The decrement operation takes two arguments, a pointer and a function, and works as follows. In the case the object pointed to (call it `x`) has a reference count of 1 before the decrement, the function executes with `x` as an argument. The function treats `x` linearly and deallocates it before it completes. It may also recursively decrement the reference counts of any subcomponents. In the other case, when `x` has a reference count greater than 1, the reference count is simply decremented and the function is not called; `unit` is returned in its place.

The main typing invariant in this system is that whenever a reference-

Syntax

`q ::=`		*qualifiers:*
	`...`	*as before*
	`rc`	*ref. count*
`t ::=`		*terms:*
	`...`	*as before*
	`inc(t)`	*increment count*
	`dec(t,t)`	*decrement count*

Qualifier Relations

$$\texttt{rc} \sqsubseteq \texttt{un} \qquad \text{(Q-RCUn)}$$

$$\texttt{lin} \sqsubseteq \texttt{rc} \qquad \text{(Q-LinRC)}$$

Typing $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{rc P}}{\Gamma \vdash \texttt{inc(t)} : \texttt{lin (rc P * rc P)}} \qquad \text{(T-Inc)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{rc P} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{lin P} \to \texttt{unit}}{\Gamma \vdash \texttt{dec(t}_1\texttt{,t}_2\texttt{)} : \texttt{unit}} \qquad \text{(T-Dec)}$$

Figure 2-16: Linear lambda calculus: Reference counting syntax and typing

counted variable appears in the static type-checking context, there is one dynamic reference count associated with it. Linear typing will ensure the number of references to an object are properly preserved.

The new `rc` qualifier should be treated in the same manner as the linear qualifier when it comes to context splitting. In other words, a reference-counted variable should be placed in exactly one of the left-hand context or the right-hand context (not both). In terms of containment, the `rc` qualifier sits between unrestricted and linear qualifiers: A reference-counted data structure may not be contained in unrestricted data structures and may not contain linear data structures. Figure 2-16 presents syntax, appropriate qualifier relation and typing rules for our reference counting additions.

In order to define the execution behavior of reference-counted data structures, and later prove that execution is sound, we will define a new sort of stored value with the form `rc(n,X) w`. The integer `n` is the reference count: it keeps track of the number of pointers the value. Given a mapping `x`$\mapsto$`rc(n,X) w`, the meta-variable `X` represents a list of `n` pseudonyms for `x`. More specifically, rather than having `x` show up `n` times in other parts of the store, each of the variables in the list `X` will show up exactly once. This arrangement facilitates the proof of type safety, which we will discuss in a moment.

The operational semantics for the new commands and for reference-counted pairs and functions are summarized in Figure 2-17. Several new bits of notation show up here to handle the relatively complex computation that must go on to increment and decrement reference counts. First, in a slight abuse of notation, we allow `q` to range over static qualifiers `un`, `lin` and `rc` as well as dynamic qualifiers un, `lin` and `rc(n,X)`. Context will disambiguate the two different sorts of uses. Second, we extend the notation $\mathtt{S}\overset{\mathtt{q}}{\sim}\mathtt{x}$ so that `q` may be `rc(n,X)` as well as `lin` and un. If `X` is the singleton set `{x}` then $\mathtt{S}\overset{\mathtt{rc(n,X)}}{\sim}\mathtt{x}$ removes the binding `y↦rc(n,X) w` from `S` when `x∈X`. Otherwise, $\mathtt{S}\overset{\mathtt{rc(n,X)}}{\sim}\mathtt{x}$ repaces the binding `y↦rc(n,X) w` with `y↦rc(n,Y-{x}) w` Finally, to increment a set of reference counts and add new pseudonyms for the appropriate pointers, we define the function `incr(S;X)`.

```
incr(S;·)      =   (S;·)
incr(S;x,X)    =   (S″;x′,X′)
                   if
                   incr(S;X) = (S′;X′)
                   S′ = S′₁,y↦rc(n,Y) w,S′₂
                   x∈Y
                   x′∉FV(S′)
                   S″ = S′₁,y↦rc(n+1;x′,Y) w,S′₂
incr(S;x,X)    =   (S′;X′)
                   if
                   incr(S;X) = (S′;X′)
                   S′(x) = un w
incr(S;X)          undefined otherwise
```

To understand how the reference counting operational semantics works, we will focus on the rules for pairs. Allocation and use of linear and unrestricted pairs stays unchanged from before as in rules (E-PAIR′) and (E-SPLIT′). Rule (E-PAIRRC) specifies that allocation of reference-counted pairs is similar to allocation of other data, except for the fact that the dynamic reference count must be initialized to 1. Use of reference-counted pairs is identical to use of other kinds of pairs when the reference count is 1: We remove the pair from the store via the function $\mathtt{S}\overset{\mathtt{rc(n,X)}}{\sim}\mathtt{x}$ as shown in rule and substitute the two components of the pair in the body of the term as shown in (E-SPLIT′). When the reference count is greater than 1, rule (E-SPLITRC) shows there are additional complications. More precisely, if one of the components of the pair, say $\mathtt{y}_1$, is reference-counted then $\mathtt{y}_1$'s reference count must be increased by 1 due to the fact that $\mathtt{y}_1$'s container is not deallocated and an additional copy of $\mathtt{y}_1$ is substituted through the body of `t`. We use the `incr` function to handle the possible increase. In most respects, the operational rules for reference-counted functions follow the same principles as reference-counted pairs. Increment and decrement operations are also relatively straightforward.

The main trickiness in carrying out the proof of safety for the language is finding store typing rules that enforce the appropriate properties and are preserved at each step in the computation. In anticipation of the difficulties, we have set up the operational semantics so that reference-counted values carry with them a set `X` of pseudonyms for the actual pointer `x` in the store binding $\mathtt{x}{\mapsto}\mathtt{rc(n,X)\ w}$. These pseudonyms each have one occurrence somewhere else in the store. This arrangement allows us to maintain our invariant that a context Γ has at most one occurence of any variable and yet place `n` assumptions that do not obey weakening or contraction laws in the context. In summary, the rule for typing reference-counted values can be the following.

$$\frac{\vdash \mathtt{S} : \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{rc\,w} : \mathtt{T}}{\vdash \mathtt{S}, \mathtt{x} \mapsto \mathtt{rc(n,x_1,\ldots,x_n)\ w} : \Gamma_2, \mathtt{x_1}{:}\mathtt{T}, \ldots, \mathtt{x_n}{:}\mathtt{T}} \quad \text{(T-NextRCS)}$$

Note that we require that none of the variables $\mathtt{x_1}$, ..., $\mathtt{x_n}$ be repeated either in the domain of the store or in `Y` when `Y` appears in other reference-counted data structures (`rc(n,Y) w`).

2.3.7 Exercise [★★,↛]: State and prove progress and preservation for the simply-typed linear lambda calculus (functions and pairs) with reference counting. □

The list of pseudonyms is really just an artifact of our proof of safety; it is not necessary for a practical implementation to mimic this list as programs actually execute. We could justify this claim after proving type safety by creating a second operational semantics in which reference-counted values have the form `rc(n) w`. Then, we might show that this second operational semantics executes in an identical (modulo the pseudonyms), step-by-step fashion to the first.

2.4 An Ordered Type System

Just as linear type systems provide a foundation for managing memory allocated on the heap, *ordered* type systems provide a foundation for managing memory allocated on the stack. The central idea is that by controlling the exchange property, we are able to guarantee that certain values, those values allocated on the stack, are used in a first-in/last-out order.

To formalize this idea, we will organize the store into two parts: a stack, which is a sequence of locations that can be accessed on one end (the "top"), and a heap, which is just like the store described in previous sections of this chapter. Pairs, functions and other objects introduced with unrestricted or linear qualifiers will be allocated on the heap as before. And as before, when a linear pair or function is used, it is deallocated. In contrast, pairs and functions introduced using an ordered qualifier (`ord`) are allocated at the top of

v ::=		*values:*
	$\ldots$	*as before*
	$rc(n,X)\ w$	*ref-counted value*
E ::=		*evaluation contexts:*
	$\ldots$	*as before*
	$inc(E)$	*inc context*
	$dec(E,t)$	*dec context*
	$dec(x,E)$	*dec context*

Evaluation $\boxed{(S;t) \longrightarrow_\beta (S';t')}$

$$\frac{(q \in \{un, lin\})}{(S;q\,\texttt{<}t_1,t_2\texttt{>}) \longrightarrow_\beta (S, x \mapsto q\,\texttt{<}t_1,t_2\texttt{>};x)} \quad \text{(E-PAIR')}$$

$$(S;rc\,\texttt{<}t_1,t_2\texttt{>}) \longrightarrow_\beta (S, x \mapsto rc(1,y)\,\texttt{<}t_1,t_2\texttt{>};y) \quad \text{(E-PAIRRC)}$$

$$\frac{S(x) = q\,\texttt{<}y_1,z_1\texttt{>} \qquad (q \in \{un, lin, rc(1,X)\})}{(S;split\ x\ as\ y,z\ in\ t) \longrightarrow_\beta (S \overset{q}{\sim} x;[y \mapsto y_1][z \mapsto z_1]t)} \quad \text{(E-SPLIT')}$$

$$\frac{S(x) = rc(n,X)\,\texttt{<}y_1,z_1\texttt{>} \qquad (n > 1) \qquad incr(S;y_1,z_1) = (S';y_1',z_1')}{(S;split\ x\ as\ y,z\ in\ t) \longrightarrow_\beta ((S' \overset{q}{\sim} x);[y \mapsto y_1'][z \mapsto z_1']t)} \quad \text{(E-SPLITRC)}$$

$$\frac{(q \in \{un, lin\})}{(S;q\,\lambda y{:}T.t) \longrightarrow_\beta (S, x \mapsto q\,\lambda y{:}T.t;x)} \quad \text{(E-FUN')}$$

$$(S;rc\,\lambda y{:}T.t) \longrightarrow_\beta (S, x \mapsto rc(1,z)\,\lambda y{:}T.t;z) \quad \text{(E-FUNRC)}$$

$$\frac{S(x_1) = q\,\lambda y{:}T.t \qquad (q \in \{un, lin, rc(1,X)\})}{(S;x_1\ x_2) \longrightarrow_\beta (S \overset{q}{\sim} x_1;[y \mapsto x_2]t)} \quad \text{(E-APP')}$$

$$\frac{S(x_1) = rc(n,X)\,\lambda y{:}T.t \qquad (n > 1 \text{ and } X = FV(\lambda y{:}T.t)) \qquad incr(S;X) = (S';X')}{(S;x_1\ x_2) \longrightarrow_\beta (S \overset{q}{\sim} x_1;[X \mapsto X'][y \mapsto x_2]t)} \quad \text{(E-APPRC)}$$

$$\frac{(x_i \in X) \qquad S{=}S_1, x \mapsto rc(n,X)\ w, S_2 \qquad S'{=}S_1, x \mapsto rc(n{+}1,(x_{n+1},X))\ w, S_2}{(S;inc(x_i)) \longrightarrow_\beta (S';lin\,\texttt{<}x_i,x_{n+1}\texttt{>})} \quad \text{(E-INC)}$$

$$\frac{(x_i \in X) \qquad (n > 1) \qquad S{=}S_1, x \mapsto rc(n,X)\ w, S_2 \qquad S'{=}S_1, x \mapsto rc(n{-}1,X{-}x_i)\ w, S_2}{(S;dec(x_i,x_f)) \longrightarrow_\beta (S';un\ ())} \quad \text{(E-DEC1)}$$

$$\frac{S{=}S_1, x \mapsto rc(1,x_i)\ w, S_2 \qquad S'{=}S_1, x \mapsto lin\ w, S_2}{(S;dec(x_i,x_f)) \longrightarrow_\beta (S';x_f\ x)} \quad \text{(E-DEC2)}$$

Figure 2-17: Linear lambda calculus: Reference counting operational semantics

the stack. Due to the lack of the exchange property, an ordered object can only be used when it is at the top of the stack. When this happens, the ordered object is popped off the top of the stack.

Syntax

The overall structure and mechanics of the ordered type system are very similar to the linear type system developed in previous sections. Figure 2-18 presents the syntax for the simply typed system. One key change from our linear type system is that we have introduced an explicit sequencing operation `let x = `t_1` in `t_2 that first evaluates the term t_1, binds the result to `x`, and then continues with the evaluation of t_2. This sequencing construct gives programmers explicit control over the order of evaluation of terms, which is crucial now that we are introducing data that must be used in a particular order. Terms that normally can contain multiple nested subexpressions such as pair introduction and function application are syntactically restricted so that their primary subterms are variables and the order of evaluation is clear. To understand why this restriction is important for the ordered type system to come, see exercise 2.4.2.

The other main addition is a new qualifier `ord` that marks data allocated on the stack. Ordered assumptions will be tracked in the type checking context Γ like other assumptions. However, they will not be subject to the exchange property. Moreover, the order that they appear in Γ mirrors the order that they will appear on the stack, with the rightmost position in Γ representing the top of the stack.

Typing

The first step in the development of the type system is the definition of rules for splitting contexts with ordered assumptions ($\Gamma = \Gamma_1 \circ \Gamma_2$). As before, we will allow unrestricted assumptions to be used in all subexpressions, and linear assumptions to be used in exactly one subexpression. Ordered assumptions should be used in exactly one subexpression, and must be used in the order in which they appear. Therefore, some (nondeterministically chosen) sequence of ordered assumptions taken from the left-hand side of Γ will be placed in Γ_1 and the remaining ordered assumptions will be placed in Γ_2. The context Γ_2 will be used by the first subexpression to be evaluated (since the top of the stack is at the right) and Γ_1 will be used by the second subexpression to be evaluated. Formally, we define the "$=$" relation in terms of two subsidiary relations, "$=_1$," which places ordered assumptions in Γ_1, and "$=_2$," which places ordered assumptions in Γ_2.

Syntax

`q` ::=		*qualifiers:*
	`ord`	*ordered*
	`lin`	*linear*
	`un`	*unrestricted*
`t` ::=		*terms:*
	`x`	*variable*
	`q b`	*Boolean*
	`if t then t else t`	*conditional*
	`q <x,y>`	*pair*
	`split t as x,y in t`	*split*
	`q λx:T.t`	*abstraction*
	`x y`	*application*
	`let x = t in t`	*sequencing*
`P` ::=		*pretypes:*
	`Bool`	*booleans*
	`T*T`	*pairs*
	`T→T`	*functions*
`T` ::=		*types:*
	`q P`	*qualified pretype*
Γ ::=		*contexts:*
	$\emptyset$	*empty context*
	$\Gamma, \texttt{x:T}$	*term variable binding*

Figure 2-18: Ordered lambda calculus: Syntax

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\frac{\Gamma =_2 \Gamma_1 \circ \Gamma_2}{\Gamma = \Gamma_1 \circ \Gamma_2} \quad \text{(M-TOP)}$$

$$\emptyset =_1 \emptyset \circ \emptyset \quad \text{(M-EMPTY)}$$

$$\frac{\Gamma =_1 \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:ord P} =_1 (\Gamma_1, \texttt{x:ord P}) \circ \Gamma_2} \quad \text{(M-ORD1)}$$

$$\frac{\Gamma =_2 \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:ord P} =_2 \Gamma_1 \circ (\Gamma_2, \texttt{x:ord P})} \quad \text{(M-ORD2)}$$

$$\frac{\Gamma =_1 \Gamma_1 \circ \Gamma_2}{\Gamma =_2 \Gamma_1 \circ \Gamma_2} \quad \text{(M-1TO2)}$$

$$\frac{\Gamma =_{1,2} \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:lin P} =_{1,2} (\Gamma_1, \texttt{x:lin P}) \circ \Gamma_2} \quad \text{(M-LINA)}$$

$$\frac{\Gamma =_{1,2} \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:lin P} =_{1,2} \Gamma_1 \circ (\Gamma_2, \texttt{x:lin P})} \quad \text{(M-LINB)}$$

$$\frac{\Gamma =_{1,2} \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:un P} =_{1,2} (\Gamma_1, \texttt{x:un P}) \circ (\Gamma_2, \texttt{x:un P})} \quad \text{(M-UN)}$$

Figure 2-19: Ordered Context Manipulation Rules

The second step is to determine the containment rules for ordered data structures. Previously, we saw that if an unrestricted object can contain a linear object, a programmer can write functions that duplicate or discard linear objects, thereby violating the central invariants of the system. A similar situation arises if linear or unrestricted objects can contain stack objects; in either case, the stack object might be used out of order, after it has been popped off the stack. The qualifier relation $\texttt{q}_1 \sqsubseteq \texttt{q}_2$, which specifies that $\texttt{ord} \sqsubseteq \texttt{lin} \sqsubseteq \texttt{un}$, *Typing* $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{un}\ (\Gamma_1, \Gamma_2)}{\Gamma_1, \mathtt{x{:}T}, \Gamma_2 \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-OVar)}$$

$$\frac{\mathtt{un}\ (\Gamma)}{\Gamma \vdash \mathtt{q\ b} : \mathtt{q\ Bool}} \quad \text{(T-OBool)}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash \mathtt{t_1} : \mathtt{q\ Bool} \\ \Gamma_1 \vdash \mathtt{t_2} : \mathtt{T} \qquad \Gamma_1 \vdash \mathtt{t_3} : \mathtt{T}\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{if}\ \mathtt{t_1}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3} : \mathtt{T}} \quad \text{(T-OIf)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \mathtt{x_1} : \mathtt{T_1} \qquad \Gamma_2 \vdash \mathtt{x_2} : \mathtt{T_2} \\ \mathtt{q(T_1)} \qquad \mathtt{q(T_2)}\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{q\ <x_1,x_2>} : \mathtt{q\ (T_1{*}T_2)}} \quad \text{(T-OPair)}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash \mathtt{t_1} : \mathtt{q\ (T_1{*}T_2)} \\ \Gamma_1, \mathtt{x{:}T_1}, \mathtt{y{:}T_2} \vdash \mathtt{t_2} : \mathtt{T}\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{split}\ \mathtt{t_1}\ \mathtt{as}\ \mathtt{x,y}\ \mathtt{in}\ \mathtt{t_2} : \mathtt{T}} \quad \text{(T-OSplit)}$$

$$\frac{\mathtt{q}(\Gamma) \qquad \Gamma, \mathtt{x{:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{q}\ \lambda \mathtt{x{:}T_1.t_2} : \mathtt{q\ T_1{\to}T_2}} \quad \text{(T-OAbs)}$$

$$\frac{\Gamma_1 \vdash \mathtt{x_1} : \mathtt{q\ T_{11}{\to}T_{12}} \qquad \Gamma_2 \vdash \mathtt{x_2} : \mathtt{T_{11}}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{x_1\ x_2} : \mathtt{T_{12}}} \quad \text{(T-OApp)}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash \mathtt{t_1} : \mathtt{T_1} \\ \Gamma_1, \mathtt{x{:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2}\end{array}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{let}\ \mathtt{x} = \mathtt{t_1}\ \mathtt{in}\ \mathtt{t_2} : \mathtt{T_2}} \quad \text{(T-OLet)}$$

Figure 2-20: Ordered lambda calculus: Typing

ensures such problems do not arise.

2.4.1 Exercise [★]: Specify the appropriate relation for polymorphic qualifiers in this ordered type system. □

The typing rules for the ordered lambda calculus appear in Figure 2-20. For the most part, the containment rules and context splitting rules encapsulate the tricky elements of the type system. Let's examine the rules for functions to see how they do it. The rule for introducing functions (T-OAbs) looks ordinary: It adds an assumption `x` to the right-hand side of the context for the purposes of checking the function body. If the function has ordered type and the argument has ordered type, this rule states that any argument passed to the function will be allocated on the top of the stack. Moreover, ordered variables in the function closure will appear deeper on the stack than the argument. In the rule for function application (T-OApp), the context is split, with the right hand side (top of the stack) being used in formation of the argument and the left-hand side being used in formation of the function. So indeed, when the function is called, its argument will appear at the top of the stack, above the objects in the function closure. The rules for booleans and pairs are designed in a similar fashion.

As a simple example, consider the following function. It takes a boolean and a pair as an argument, allocated sequentially at the top of the stack. The boolean is at the very top of the stack and the integer pair is next (the top of the stack is to the left). If the boolean is true, it leaves the components of the pair in the same order as they appeared on input and otherwise, it swaps their order.

```
lamda x:ord (ord (int * int) * bool).
  split x as p,b in
  if b then
    p
  else
    split p as i1,i2 in
    ord <i2,i1>
```

2.4.2 EXERCISE [★,RECOMMENDED]: Write a program that demonstrates what can happen if the syntax of pair formation is changed to allow programmers to write nested subexpressions (ie: we allow `ord <`e_1`,`e_2`>` rather than `ord <x,y>`)? □

The stack typing discipline that arises from our ordered type system is not too flexible as it does not allow programmers to use objects allocated deep on the stack; all uses must occur at the top of the stack. However, this is exactly the sort of typing discipline required to check that the operation of stack-based virtual machines such as the JVM is safe.

2.4.3 EXERCISE [★★★,$\not\rightarrow$]: Look up the definition of the Java Virtual Machine Language (Yellin (**?**) provides a good reference) and design an ordered type system to keep track of the machine operand stack. □

Another related application of this ordered type system involves type-preserving compilation of functional languages via the *continuation-passing style* (CPS) transform. CPS decomposes ordinary high-level functions into low-level CPS functions that explicitly take a *continuation* (the function's return address, itself a function) as an argument and explicitly invoke (call) the continuation to return, the return value being the argument of the continuation. For example, we can recode the function above in CPS as follows.

```
type cont = ord (ord (int * int) → ans)

lamda x:ord (cont * ord (bool * ord (int * int))).
  split x as cont,x in
  split x as p,b in
  if b then
    cont p
  else
    split p as i1,i2 in
    cont (ord <i2,i1>)
```

We had to make two main changes to the function. First, we split the incoming argument into two parts: the actual argument and the continuation. Second, rather than returning the newly allocated pair, we call the continuation. We assume the type `ans` is the type of the final result of the computation.

Since the translation changes the calling convention for functions, the translation must also change function application. Each function call with the form `f x` will now have the form `f ord <k,x>` where `k` is the current continuation.

2.4.4 EXERCISE [★★★★, ↛]: Research the continuation-passing style transformation in the literature. Define a type-preserving, continuation-passing style translation from the ordinary, simply-typed lambda calculus to the ordered lambda calculus that allocates continuations on the stack. State and prove a theorem that shows your translation this type-preserving. A basis for the solution can be found in research on ordered type theory by Polakow and Pfenning (**?**). □

Operational Semantics

To define the operational semantics for our new ordered type system, we will divide our previous stores into two parts, a heap `H` and a stack `K`. Both are just a list of bindings as stores were before (see Figure 2-21 for a summary).

Rather than rewrite our operational semantics for the new ordered (stack-allocated) elements, we simply alter the definition of our store manipulation functions. First, we need to say what it means to add a binding to the store. This is straightforward: unrestricted and linear bindings are added to the heap and ordered bindings are added to the stack.

$$\begin{array}{lcl} (\mathtt{H};\mathtt{K}),\mathtt{x}\mapsto \mathtt{ord}\,\mathtt{w} & = & (\mathtt{H};\mathtt{K},\mathtt{x}\mapsto \mathtt{ord}\,\mathtt{w}) \\ (\mathtt{H};\mathtt{K}),\mathtt{x}\mapsto \mathtt{lin}\,\mathtt{w} & = & (\mathtt{H},\mathtt{x}\mapsto \mathtt{lin}\,\mathtt{w};\mathtt{K}) \\ (\mathtt{H};\mathtt{K}),\mathtt{x}\mapsto \mathtt{un}\,\mathtt{w} & = & (\mathtt{H},\mathtt{x}\mapsto \mathtt{un}\,\mathtt{w};\mathtt{K}) \end{array}$$

Second, we need to specify how to remove a binding from the store.

$$\begin{array}{lcl} (\mathtt{H};\mathtt{K}_1,\mathtt{x}\mapsto \mathtt{v},\mathtt{K}_2) \overset{\mathtt{ord}}{\sim} \mathtt{x} & = & \mathtt{H};\mathtt{K}_1,\mathtt{K}_2 \\ (\mathtt{H}_1,\mathtt{x}\mapsto \mathtt{v},\mathtt{H}_2;\mathtt{K}) \overset{\mathtt{lin}}{\sim} \mathtt{x} & = & \mathtt{H}_1,\mathtt{H}_2;\mathtt{K} \\ (\mathtt{H};\mathtt{K}) \overset{\mathtt{un}}{\sim} \mathtt{x} & = & \mathtt{H};\mathtt{K} \end{array}$$

With these simple changes, the evaluation rules from previous sections can be reused essentially unchanged. We only need to add the the evaluation context for sequencing (`let x = E in t`) and its evaluation rule.

$\mathtt{S}$ ::=		*stores:*
	$\mathtt{H;K}$	*complete store*
$\mathtt{H}$ ::=		*heap:*
	$\emptyset$	*empty context*
	$\mathtt{H, x \mapsto v}$	*heap binding*
$\mathtt{K}$ ::=		*stack:*
	$\emptyset$	*empty context*
	$\mathtt{K, x \mapsto v}$	*stack binding*

Figure 2-21: Ordered Lambda Calculus: Operational Semantics

$$(\mathtt{S;let\ x = x_1\ in\ t_2}) \longrightarrow_\beta (\mathtt{S;}[\mathtt{x} \mapsto \mathtt{x_1}]\mathtt{t_1}) \qquad \text{(E-LET)}$$

In all cases but one, deallocation of an ordered object will occur at the top of the stack. In each of these cases, the ordered deallocation operation could be rewritten as follows and implemented as an efficient stack pointer increment.

$$(\mathtt{H;K,x \mapsto v}) \overset{\mathtt{ord}}{\sim} \mathtt{x} \;=\; \mathtt{H;K}$$

The single anomalous case is the case for application of an ordered function to an ordered argument. In this case, the stack is organized with the ordered variables in the function appearing lowest on the stack, the ordered function pointer appearing immediately above that and the ordered argument appearing on top. However, the ordered function pointer is used (and deallocated) immediately upon application, whereas the ordered argument may be used at some later point in the function body. Hence, to call an ordered function with an ordered argument, a low-level implementation would have to first load the function pointer from its position on the stack, then shift the argument down the stack so it occupies the space directly above the variables in the ordered function's closure, and finally, begin to run the ordered function's code.

2.4.5 EXERCISE [★★★]: Define new typing rules and a new operational semantics specific to the case when an ordered function is applied to an ordered argument. Come up with a system that avoids having to "shift" the argument down the stack when the function is called. □

2.4.6 EXERCISE [★★,$\nrightarrow$]: It would be nice if we could access objects deep on the stack. Define the static and dynamic semantics of an operation that copies the data $\mathtt{n}$ locations from the top of the stack into the heap (for any abitrary, but statically known integer $\mathtt{n}$). □

2.5 Further Applications

Memory management applications make good motivation for substructural type systems and provides a concrete framework for studying their properties. However, substructural types systems, and their power to control the number and order of uses of data and operations, have found many applications outside of this domain. In the following paragraphs, we informally discuss a few of them.

Controlling Temporal Resources

We have studied several ways that substructural type systems can be used to control physical resources such as memory and files. What about controlling the temporal resources? Amazingly, substructural type systems can play a role here as well: Careful crafting of a language with an *affine* type system, where values are used at most once, can ensure that computations execute in polynomial time.

To be begin, we will allow our polynomial time language to contain affine booleans, pairs and (non-recursive) functions. In addition, to make things interesting, we will add affine lists to our language, which have constructors `nil` and `cons` and a special iterator to recurse over objects with list type. Such iterators have the following form.

$$\texttt{iter (stop} \Rightarrow \texttt{t}_1 \mid \texttt{x with y} \Rightarrow \texttt{t}_2\texttt{)}$$

If $\texttt{t}_1$ and $\texttt{t}_2$ have type `T`, our iterator defines a function from lists to objects with type `T`. Operationally, the iterator does a case to see whether its input list is `nil` or `cons(hd,tl)` and executes the corresponding branch. More specifically we can define the operation of iterators using two simple rules.[3]

$$\texttt{iter (stop} \Rightarrow \texttt{t}_1 \mid \texttt{hd with rest} \Rightarrow \texttt{t}_2\texttt{) nil} \longrightarrow_\beta \texttt{t}_1 \qquad \text{(E-IterNil)}$$

$$\frac{\texttt{iter (stop} \Rightarrow \texttt{t}_1 \mid \texttt{hd with rest} \Rightarrow \texttt{t}_2\texttt{) v}_2 \longrightarrow_\beta * \texttt{v}_2'}{\begin{array}{c}\texttt{iter (stop} \Rightarrow \texttt{t}_1 \mid \texttt{hd with rest} \Rightarrow \texttt{t}_2\texttt{) cons(v}_1\texttt{,v}_2\texttt{)} \longrightarrow_\beta \\ [\texttt{hd} \mapsto \texttt{v}_1][\texttt{rest} \mapsto \texttt{v}_2']\texttt{t}_2\end{array}} \qquad \text{(E-IterCons)}$$

In the second rule, the iterator is invoked inductively on `tl`, giving the result $\texttt{v}_2'$, which is used in term $\texttt{t}_2$.

The `append` function below illustrates the use of iterators.

3. Since we are not interested in memory management here, we have simplified our operational semantics from previous parts of this chapter by deleting the explicit store and using substitution instead. The operational judgment has the form $\texttt{t} \longrightarrow_\beta \texttt{t}'$ and, in general, is defined similarly to the operational systems in *Types and Programming Languages*.

```
val append : T list→T list→T list =
  iter (
    stop ⇒ λ(l:T list).l
  | hd with rest ⇒ λ(l:T list).cons(hd,rest l))
```

When applied to a list l_1, the iterator builds up a function that expects a second list l_2 and concatenates l_2 to the end of l_1. Clearly, `append` is a polynomial time function, a linear-time one in fact, but it is straightforward to write exponential time algorithms in the language as we have defined it so far. For instance:

```
val double : T list→T list =
  iter (
    stop ⇒ nil
  | hd with rest ⇒ cons(hd,cons(hd,rest)))

val exp : T list→T list =
  iter (
    stop ⇒ nil
  | hd with rest ⇒ double (cons(hd,rest)))
```

The key problem here is that it is trivial to write iterators like `double` that increase the size of their arguments. After constructing such an iterator, we can use it as the inner loop of another iterator, like `exp`, and cause an exponential blow-up in our running time. However, this is not the only problem. Higher-order functions make it even easier to construct exponential-time algorithms:

```
val compose =
  λ(fg:(T list→T list) * (T list→T list)).
    λ(x:T list).
      split fg as f,g in f (g x)

val junk : T

val exp₂ : T list→T list→T list =
  iter (
    stop ⇒ λ(l:T list).cons(junk,l)
  | hd with rest ⇒ λ(l:T list).compose <rest,rest> l)
```

Fortunately, a substructural type system can be used to eliminate both problems by allowing us to define a class of *non-size-increasing* functions and by preventing the construction of troublesome higher-order functions, such as $\texttt{exp}_2$.

The first step is to demand that all user-defined objects have affine type. They can be used zero or one times, but not more. This restriction immediately rules out programs such as `exp`$_2$. System defined operators like cons can be used many times.

The next step is to put mechanisms in place to prevent iterators from increasing the size of their inputs. This can be achieved by altering the cons constructor so that it can only be applied when it has access to a special resource with type `R`.

```
operator cons : (R,T,T list) → T list
```

There is no constructor for resources with type `R` so they cannot be generated out of thin air; we can only apply `cons` as many times as we have resources. We also adapt the syntax for iterators as follows.

```
iter (
  stop ⇒ t₁
| hd with tl and r ⇒ t₂)
```

Inside the second clause of the iterator, we are only granted a single resource (`r`) with which to allocate data. Consequently, we can allocate at most one cons cell in `t`$_2$. This provides us with the power to rebuild a list of the same size, but we cannot write a function such as `double` that doubles the length of the list or `exp` that causes an exponential increase in size. To ensure that a single resource from an outer scope does not percolate inside the iterator and get reused on each iteration of the loop, we require that iterators be closed, mirroring the containment rules for recursive functions defined in earlier sections of this chapter.

Although restricted to polynomial time, our language permits us to write many useful functions in a convenient fashion. For instance, we can still write append much like we did before. The resource we acquire from destructing the list during iteration can be used to rebuild the list later.

```
val append : T list → T list → T list =
  iter (
    stop ⇒ λ(l:T list).l
  | hd with rest and r ⇒ λ(l:T list). cons(r,hd,rest l))
```

We can also write `double` if our input list comes with appropriate credits, in the form of unused resources.

```
val double : (T*R) list → T list =
  iter (
    stop ⇒ nil
  | hd with rest and r1 ⇒
     split hd as x,r2 in cons(r1,hd,cons(r2,hd,rest)))
```

Fortunately, we will never be able to write `exp`, unless, of course, we are given an exponential number of credits in the size of the input list. In that case, our function `exp` would still only run in linear time with respect to our overall input (list and resources included).

The proof that all (first-order) functions we can define in this language run in polynomial time uses some substantial domain theory that lies outside the scope of this book. However, the avid reader should see Section 2.6 for references to the literature where these proofs can be found.

Compiler Optimizations

Many compiler optimizations are enabled when we know that there will be *at most one use* or *at least one use* of a function, expression or data structure. If there is at most one use of an object then we say that object has *affine* type. If there is at least one use then we say the object has *relevant* (or *strict*) type. Here are just a few optimizations that usage information can enable.

- *Floating in bindings*. Consider the expression `let x = e in (λy....x...)`. Assume the computation `e` has no effects, is it a good idea to float the binding inside the lambda and create the new expression `λy.let x = e in (...x...)`? The answer depends on how many times the resulting function is used. If it is used at most once, the optimization is probably a good one: we may avoid computing `e` and will never compute it more than once.

- *Inlining expressions*. In the example above, if we have the further information that `x` itself is used at most once inside the body of the function, then we might want to substitute the expression `e` for `x`. This may give rise to further local optimizations at the site where `e` is used. Moreover, if it turns out that `e` is used zero times (as opposed to one time) we will have saved ourselves the trouble of computing it.

- *Thunk update avoidance*. In lazy functional languages such as Haskell, evaluation of function parameters is delayed until the parameter is actually used in the function body. In order to avoid recomputing the value of the parameter each time it is used, implementers make each parameter a *thunk* — a reference that may either hold the computation that needs to be run or the value itself. The first time the thunk is used, the computation will be run and will produce the necessary result. In general, this result is stored back in the thunk for all future uses of the parameter. However, if the compiler can determine that the data structure is used as most once, this thunk update can be avoided.

- *Eagerness*. Again in lazy functional languages, if we can guarantee that an object is used at least once, then we can evaluate it right away and avoid creating a thunk altogether.

The optimizations described above may be implemented in two phases. The first phase is a program analysis that may be implemented as affine and/or relevant type inference. After the analysis phase, the compiler uses the information to transform programs. Formulating compiler optimizations as type inference followed by type-directed translation has a number of advantages over other techniques. First, the language of types can be used to communicate optimization information across modular boundaries. This can facilitate the process of scaling intra-procedural optimizations to inter-procedural optimizations. Second, the type information derived in one optimization pass can be maintained and propagated to future optimization passes or into the back end of the compiler where it can be used to generate Typed Assembly Language or Proof-Carrying Code, as discussed in Chapters ♠**??** and ♠**??**.

2.6 Notes

Substructural logics are very old and date back to at least Orlov (**?**), who axiomatized the implicational fragment of relevant logic. Somewhat later, Moh (**?**) and Church (**?**) provided alternative axiomatizations of the relevant logic now known as R. In the same time period, Church was developing his theory of the lambda calculus at Princeton University, and his λI calculus (**?**), which dissallowed abstraction over variables that did not appear free in the body of the term, was the first substructural lambda calculus. Lambek (**?**) introduced the first "ordered logic," and used it to reason about natural language sentence structure. More recently, Girard (**?**) developed linear logic, which gives control over both contraction and weakening, and yet provides the full power of intuitionistic logic through the unrestricted modality "!". O'Hearn and Pym (**?**) show that the logic of bunched implications provides another way to recapture the power of intuitionistic logic while giving control over the structural rules.

For a comprehensive account of the history of substructural logics, please see Došen (**?**), who is credited with coining the phrase "substructural logic," or Restall (**?**). Restall's textbook on substructural logics (**?**) provides good starting point to those looking to study the technical details of either the proof theory or model theory for these logics.

John Reynolds pioneered the study of substructural type systems for programming languages with his development of syntactic control of interference (**??**), which prevents two references from being bound to the same variable and thereby facilitates reasoning about Algol programs. Later, Girard's development of linear logic inspired many researchers to develop functional languages with linear types. One of the main applications of these new type systems was to control effects and enable in-place update of arrays in pure functional languages.

Lafont (**?**) was the one of the first to study progamming languages with linear types; he developed a linear abstract machine. Lafont was soon followed by many other researchers, including Baker (**?**) who informally showed how to compile Lisp into a linear assembly language in which all allocation, deallocation and pointer manipulation is completely explicit, yet safe. Another influential piece of work is due to Chirimar, Gunter and Riecke (**?**) who developed an interpretation of linear logic based on reference counting. Their reference counting scheme is slightly different than the one proposed here due to the differences between linear logic and the (non-logical) linear types that make up our system. Turner and Wadler (**?**) summarize two computational interpretations that arise directly through the Curry-Howard isomorphism from Girard's linear logic. They differ from the account given in this chapter as neither account has both shared, useable data structures and deallocation. These two features, which are essential if one is to build a practical programming language, appear incompatible with a type system derived directly from linear logic and its single unrestricted modality.

The development of practical linear type systems with two classes of type, one linear and one unrestricted, began with Wadler's work (**?**) in the early nineties. The presentation given in this chapter is inspired by work from Wansbrough and Peyton Jones (**?**) and Walker and Watkins (**?**). Wansbrough and Peyton Jones included qualifier subtyping and bounded parametric polymorphism in their system in addition to many of the features described here. However, the parameterization of our system according to the combination of `un`, `lin`, and `rc` qualifiers and containment rules is novel. The idea of formulating the system with a generic context splitting operator was taken from Cervesato and Pfenning's presentation of Linear LF (**?**).

The algorithmic type system described in section 2-5 solves what is commonly known in the linear logic programming and theorem proving literature, as the *resource management problem*. Many of the ideas for the current presentation came from work by Cervasato and Pfenning (**?**), who solve the more general problem that arises when linear logic's additive connectives are considered. Hofmann takes a related approach when solving the type inference problem for a linearly-typed functional language (**?**).

The ordered type system developed here is also novel. However, it was derived from Polakow and Pfenning's ordered logic (**?**), in the same way that the practical linear type systems mentioned above emerged from linear logic. The closest related type system (as opposed to logic) that we are aware of is Petersen's ordered lambda calculus (**?**); this latter system does not classify types using qualifiers as we have done, but rather uses a single "mobility" modality.

Analysis and reasoning about the time and space complexity of programs has always been an important part of computer science. However, the use of programming language technology, and type systems in particular, to automatically constrain the complexity of programs is somewhat more recent. For instance, Bellantoni and Cook (**?**) and Leivant (**?**) developed predicative systems that control the use and complexity of recursive functions. It is possible to write all, and only, the polynomial-time functions in their system. However, it is not generally possible to compose functions and therefore many "obviously" polynomial-time algorithms cannot be coded naturally in their system. Girard (**?**), Hofmann (**??**), and Bellantoni et al. (**?**) show how linear type systems can be used to alleviate some of these difficulties. The material presented in this chapter is derived from Hofmann's work.

One of the most successful and extensive applications of substructural type systems in programming practice can be found in the Concurrent Clean programming language (**?**). Clean is a commercially developed, pure functional programming language. It uses *uniqueness types* (**?**), which are a variant of linear types, and strictness annotations (**?**) to help support concurrency, I/O and in-place update of arrays. The implementation is fast and is fully supported by a wide range of program development tools including an Integrated Development Environment for project management and GUI libraries, all developed in Clean itself.

Substructural type systems have also found gainful employment in the intermediate languages of the Glaskow Haskell Compiler. For instance, Turner et al. (**?**) and Wansbrough and Peyton Jones (**?**) showed how to use affine types and affine type inference to optimize programs as discussed earlier in this chapter. They also use extensive strictness analysis to avoid thunk creation.

Recently, researchers have begun to investigate ways to combine substructural type systems with dependent types and effect systems such as those described in Chapters ♠**??** and ♠**??**. The combination of both dependent and substructural types provides a very powerful tool for enforcing safe memory management and more general resource-usage protocols. For instance, DeLine and Fähndrich developed Vault (**??**), a programming language that uses static capabilities (**?**) (a hybrid form of linear types and effects) to enforce a variety of invariants in Microsoft Windows device drivers including locking protocols, memory management protocols and others. Cyclone (**??**), a completely type-safe substitute for C, also uses linear types and effects to grant programmers fine-grained control over memory allocation and deallocation. In each of these cases, the authors do not stick to the pure linear types described here. Instead, they add coercions to the language to allow linearly-typed objects to be temporarily aliased in certain contexts, following a long line of research on this topic (**?????**).

A *Solutions to Selected Exercises*

2.1.4 SOLUTION: The proof of each lemma proceeds by induction on the typing derivation. Almost all cases follow directly from the induction hypothesis. The base cases are straightforward as well, but some slight amount of work is involved. For instance, in the base case for weakening we are given the judgement Γ_1, `x:T`, $\Gamma_2 \vdash$ `x : T`. and must prove that for arbitrary Γ_3, Γ_1, `x:T`, Γ_2, $\Gamma_3 \vdash$ `x : T`. The latter judgement follows directly from the variable rule as the rule schema allows the context Γ_1, `x:T`, Γ_2, Γ_3. Notice, however, that if we were not careful in the definition of the variable rule and had omitted Γ_2 from the context in the rule schema, we would be unable to prove this weakening lemma. Hence, while simple, the rules for the variables and constants play an integral role in defining the structural properties of a type system.

2.2.1 SOLUTION: Since the variable may only appear on the extreme right-hand side of the context, we will be unable to prove the exchange lemma. In the literature, you will see this formulation of the variable rule all the time because authors often treat contexts as finite partial maps. In other words, contexts that differ only in the order in which we write down their elements are treated equally and are never distinguished from one another. In this chapter, we choose not to take this perspective so that we may study the complete set of structure rules directly.

2.3.2 SOLUTION: The type of linear trees with elements of type `T` follows.

```
type T tree = rec a.lin (unit + lin (T * a * a))
```

It will be convenient to define some constructors for trees of type `T` as well.

```
fun nil_T (nil:unit) : T tree = roll (lin inl nil)

fun node_T (arg : lin (T * T tree * T tree)) : TL = roll (lin inr arg)
```

In order to do a constant-space tree traversal, we must reuse the tree cells to create our own list (stack) that remembers what to do next after we have completed processing the current subtree. In ML, we might define such a list using the datatype:

```
datatype (T1,T2) TL =
  done
| right of T2 * T1 tree * TL
| left of T2 * T2 tree * TL
```

The first constructor indicates there is nothing left to do. The second constructor indicates we have finished processing a left subtree, but we still need to process the right subtree (the object with type T_1 tree) and glue the processed tree element (with type T_2) together with the left and right subtrees. We also need to recursively process the rest of the list. The last constructor indicates we have just finished processing a right subtree and we need to assembly the tree element, left and right subtrees and recursively process the rest of the list.

In our linear lambda calculus, the ML type definition given above and its associated constructors will be defined as follows. We will use $\mathtt{in}_0$, $\mathtt{in}_1$,... $\mathtt{in}_{n-1}$ to inject into a n-ary sum when n is greater than two. For brevity, we will not bother to index these types and constructors with parameters T_1 and T_2.

```
type TL =
  mu a. lin (unit + lin (T2 * T1 tree * TL) + lin (T2 * T2 tree * TL))

fun done (nil:unit) : TL = roll (lin in0 nil)

fun left (arg : lin (T2 * T2 tree * TL)) : TL = roll (lin in1 arg)

fun right (arg : lin (T2 * T1 tree * TL)) : TL = roll (lin in2 arg)
```

The algorithm is factored into a top-level function treeMap and two helpers. The first processes a subtree we have not seen yet. The second determines what to do next by looking at the TL stack.

```
type FT = T1 ⟶ T2

fun treeMap(f:FT,t:T1 tree) : T2 tree =
  procTree (f,t,empty())

and procTree(f:FT,t:T1 tree,tl:TL) : T2 tree =
  case unroll t (
    inl nil ⇒ procTL (f,empty(),tl)
  | inr tree ⇒
     split tree as elem,t1,t2 in
     procTree (f,t1,right lin <f elem,t2,tl>)

and procTL(f:FT,t:T2 tree,tl:TL): T2 tree =
  case unroll tl (
    in0 nil  ⇒ t
  | in1 arg ⇒
      split arg as elem,t2,tl in
      procTree (f,t2,left lin <elem,t,tl>)
  | in2 arg ⇒
```

```
        split arg as elem,t1,tl in
        procTL (f,nodeT lin <elem,t1,t2>,tl)
```

2.3.5 SOLUTION: If an unrestricted array can contain a linear object, the linear object might never be used. Interestingly, due to our swapping operational semantics for arrays, even though an unrestricted array (containing linear objects) could be used many times, the linear objects themselves could never be used more than once.

2.4.1 SOLUTION: Since `ord` is the least qualifier in our ordered type system and `un` is the greatest qualifier in our ordered type system, the rules dealing with polymorphic qualifiers should be following.

$$\texttt{ord} \sqsubseteq \texttt{p} \qquad \text{(Q-OrdP)}$$

$$\texttt{p} \sqsubseteq \texttt{un} \qquad \text{(Q-PUn)}$$

2.4.2 SOLUTION: Consider the following expression. If we generalized the syntax to allow nested sub expressions but made no change to the typing rules, it would type check despite the fact that booleans are confused with integers.

```
let x = ord <true,true> in
let y = ord <ord <3,2>,x> in
split y as z1,z2 in
split z2 as b1,b2 in
if b1 then ...  (* using an int as if it was a bool *)
```

2.4.5 SOLUTION: There are undoubtedly several solutions to this exercise. The simplest solution is to add a new `junk` type that can serve as a placeholder for the used function pointer. The only thing that can be done with an object of type `junk` is to pop it off the stack. The idea is that rather than having the compiler implicitly insert instructions to shift the ordered argument down the stack at the point of a function call, the function will be left on the stack, but given the unusable type `junk`. When the function body has used the argument, the junk item will appear at the top of the stack. The programmer will have to explicitly pop it off the stack before using the objects in the function closure, which will appear deeper on the stack.

The typing rule for the specialized ordered abstraction with ordered argument as well as the typing rule for the command $\texttt{pop}\,\texttt{t}_1\texttt{;}\ \texttt{t}_2$, which pops its argument ($\texttt{t}_1$) off the top of the stack and continues execution with $\texttt{t}_2$, appear below. It is up to you to define their operational rules.

$$\frac{\texttt{q}(\Gamma) \qquad \Gamma, \texttt{f:ord junk}, \texttt{x:ord}\,\texttt{P}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{ord}\,\lambda_f\,\texttt{x:(ord}\,\texttt{P}_1\texttt{).t}_2 : \texttt{ord (ord}\,\texttt{P}_1\texttt{)}\rightarrow\texttt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{ord\ junk} \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{pop}\ \mathtt{t}_1\mathtt{;}\ \mathtt{t}_2 : \mathtt{T}} \qquad \text{(T-Pop)}$$