

Cause and effect: type systems for effects and dependencies

Geoffrey Washburn
Technical Report MS-CIS-05-???
Department of Computer and Information Science
University of Pennsylvania
geoffw@cis.upenn.edu

Sunday 6th March, 2005

Draft version: \$Id: wpeii.tex 625 2005-03-05 20:11:10Z geoffw \$
Document generated on: Sunday 6th March, 2005 at 01:06

Abstract

Formal frameworks for reasoning about programs are important not only for automated tools but also for programmers. Type systems have proven an enormously popular framework for validating static analyses of programs, as well as for documenting their interfaces for programmers. However, most type systems used in practice today fail to capture many essential aspects of program behavior: the effects and dependencies of programs. There has been considerable research in the past twenty years into developing type systems that capture this information. In this paper we examine, compare, contrast, and connect a number of highly influential and prototypical type systems for capturing effects and dependencies. Specifically we look at classic effect type systems as conceived by Gifford et.al, a canonical example of dependency type systems – type systems for information-flow, two different modal type systems that enforce a (co)monadic effect and dependency discipline, and linear type systems for precise reasoning about states and resources. Finally, we also present a calculus that provides an insight into a possibility for an unified account for all of these systems.

1 Introduction

Precise reasoning about programs is important both for automated tools and programmers. An understanding of a program's behavior allows a compiler to perform optimizations that might not otherwise be possible or allow a static analyzer to be less conservative. Likewise, descriptive signatures for application programming interfaces, or other library routines, give programmers a better understanding of how these functions will interact with their code. Just as importantly, if there is automated support for providing the programmer with feedback about the behavior of their own code, they will have a better idea of whether it meets their specifications.

Dating back as far as to Fortran [5], type specifications in statically typed languages have been able to guide tools and programmers in understanding the behavior of programs and individual functions. A typical type specification will give types to the inputs that a function expects and a type to its output. At a minimum this provides a vague form of documentation, allows a compiler to produce the correct invocation and return handlers, and provides a simple way to root out errors caused by attempting to provide a function with nonsensical input. In languages with more sophisticated type systems it is actually possible to derive properties and laws about functions from their just their types. In fact, some types are precise enough that they completely specify the possible implementations of function.

Most famously, Wadler showed how to derive »free« theorems about functions in the polymorphic λ -calculus solely from their types [44]. A relatively simple example is the functions with type $\forall \alpha . \alpha \rightarrow \alpha$. Ignoring extensions for general recursion and intensional type analysis [21], there is only one mathematical function that could implement this interface¹: the identity function. Because the type α is kept abstract, as a consequence of the parametricity property of the polymorphic λ -calculus, the only way such a function could produce an output of type α is to return the input it was given.

With even more advanced type systems, more sophisticated reasoning is possible. In a language with dependent [4] or indexed types, such as DML [47], it is possible to write types that are indexed by integer constraints. For example, the type of a list of integers that has five elements could be written as $\text{list}(5) \text{ int}$. It would then be possible to write the interface of the list map function as $\Pi i . \text{list}(i) \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{list}(i) \beta$, where Π abstracts over integer indices. This type captures the property that the output list will always be the same length as the input list. It is possible to continue pushing these ideas further to allow specifying the behavior of functions with even greater detail. However, the types we have discussed fail to capture essential behaviors in real programs.

Many realistic programs are more than just collections of idealized mathematical functions composed together. They will print information to the screen, write data to memory or external storage, communicate with other programs on the same computer or over the network, or simply fail due to an error condition. Most languages in common use today do not provide any way to write types or specify interfaces for functions and programs that capture these sorts of »effectful« behavior².

1.1 Effects

Before we begin examining type systems that capture the »effectful« behaviors of a function or program, it is worthwhile to review just what is meant by an »effect«. Informally, an effect is generally considered to be any observable behavior of a program. For example, allocation and mutation of the heap is an effect, as is printing to the screen, as is divergence and non-local control transfers. However, it is quite difficult to develop a complete taxonomy for what is and is not an effect. In actually, it appears to be much simpler to instead characterize the necessary conditions for a program to be considered *pure*, that is, free from effects.

¹There will however be an infinite collection of syntactic terms corresponding to this ideal function.

²A notable exception is Haskell [36], and in a very limited fashion, Java [20]. Arguably monads may be used in a number of languages with sophisticated type systems, but this seems to be a rarity outside of purely functional languages.

Firstly, if we imagine programs to be idealized mathematical functions, a program will only be pure if it is total on its domain. If a program is not total on its domain, this necessarily implies that there is some input we can provide the program that will not yield an answer. The fact that an answer is not provided implies that any number of »effectful« control transfers occurred: the program diverged, an exception was thrown, some other sort of non-local control transfer occurred, etc. Park and Harper [34] do not define their terminology in detail, but we believe that their notion of a *control effect* corresponds to any effect that will necessarily cause a program to be partial on its domain. There is also a close correspondence to the notion in game semantics of strategies that are not *bracketed*, those strategies where interaction does not follow a strictly nested structure [3]. It is not clear whether this accounts for divergence effects, however, because diverging programs will maintain proper nesting, but the nesting will become infinitely deep³.

Totality is not a sufficient condition for a program to be pure. A program that takes an integer input and prints it to the screen is total with respect to a concrete architecture’s arithmetic, but is something that would informally be considered to be effectful. Unfortunately, crafting a condition to cover all possible effects of this class is quite tricky. Again we think that there is a connection here with Park and Harper’s notion of a *world effect*. Abstractly we assume the existence of some model of the universe w , and for a program to be considered pure, it must not alter the model, and its result must be the same regardless of the instantiation of the model. This seems reasonable, but it means that we are restricted to a notion of purity parameterized by some model of the universe. We conjecture that the more abstract the model, the more functions there will be that appear to be pure. As with control effects, game semantics provides a characterization analogous to world effects with what are called *innocent* strategies. An innocent strategy is one that must behave the same regardless of the context in which it is run.

1.2 Dependencies

If we were to characterize an effect as something that a program causes to happen, a *dependency* is tantalizingly close to the dual notion: something that must have happened for the program to have produced its result. Knowing about the dependencies of a program or function can be just as important for a compiler or a programmer as knowing what effects it might produce. Tracking the dependencies of a program with a type system has become extremely popular in the context of language based security. In what are called information-flow type systems [48], types are annotated with information about the privilege level required for a given value to be computed. This provides programmers and tools with a means to ensure that programs do not inadvertently or maliciously choose to release privileged information.

Dependency type systems have also been used in program analyses such as slicing and binding-time analysis. Program slicing tracks the overall dependencies within a program – »the value computed by function f depends upon function g « [46]. Binding-time analysis attempts to divide computations in the program into those that can be performed statically and those that must be performed dynamically; binding-time analysis is particularly important in the context of partial-evaluation [11, 13]. Liveness analysis in optimizing compilers is another example of a dependency analysis, though it is generally not formalized within a type system.

³Furthermore, it seems unlikely that game semantics would restrict interactions to be finite, otherwise it could not provide complete models for PCF

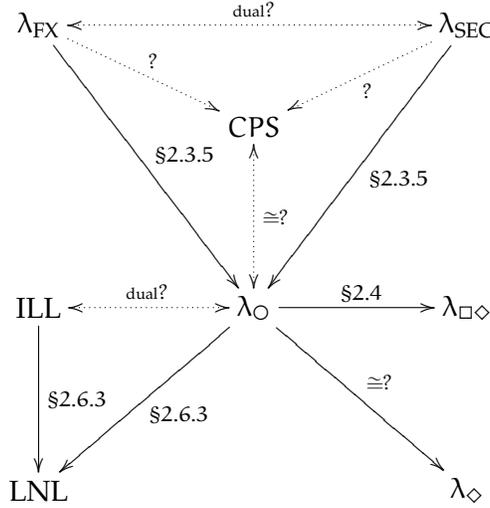


Figure 1: Known and conjectured relationships between type systems

1.3 The road-map

Given the importance of tracking effects and dependencies in programs, the problem has not surprisingly received considerable attention since the earliest optimizing compilers were written. However, we feel that much of this work can be characterized as program analyses and heuristics for discovering the effect and dependency relationships in programs, rather than formal systems for reasoning about effects and dependencies. When we limit our scope to type systems in particular, there are clearly a few specific systems that we feel have proven enormously influential.

In this paper we endeavor to review, compare, and connect what we felt were fundamental type systems for capturing effects and dependencies. Interestingly, all of these systems were developed past twenty years⁴. Figure 1 provides an overview of the calculi we will examine and known and some conjectured relationships between them:

- Section 2.1: λ_{FX} . This is an idealization of a prototypical effect type system.
- Section 2.2: λ_{SEC} . This is an idealization of a prototypical dependency type system.
- Section 2.3: λ_O . A language that tracks effects and dependencies through a monadic structure.
- Section 2.4: $\lambda_{\square\Diamond}$. A language that tracks effects and dependencies through a combined comonadic and monadic structure.
- Section 2.5: ILL. The term language for Intuitionistic Linear Logic, a logic for reasoning about resources and state.

⁴Of course, their creators were invariably standing on the shoulders of giants.

- Section 2.6: LNL. The term language for Linear/Non-linear Logic, a calculus that neatly captures ILL and λ_{\circ} as near duals.

In Section 3 we summarize our survey and discuss open problems.

2 The bestiary

In this section, we describe six different languages that are representative of the approaches that have been used to date in formalizing and reasoning about languages with effects and dependencies. The reader is assumed to have a working understanding of the simply-typed λ -calculus, type systems, operational semantics, and the associated mathematics. See a standard introductory text for more background [30, 40].

In general, we will elide the operational semantics of the calculi, as the reductions and substitutions will normally be obvious. We will assume a call-by-value operational semantics, unless otherwise noted.

Finally, to concentrate upon the essential differences in the languages, we have chosen to use a single uniform notation and structure for describing specific effects and dependencies in the remainder of this paper. We will call our syntactic category labels, as they can be used as a basis for both effects and dependencies.

Definition 2.1 *A label structure $\langle \mathcal{L}, \leq, \emptyset, \bullet \rangle$ consists of:*

- *A partially-ordered set of primitive labels, $\langle \mathcal{L}, \leq \rangle$.*
- *A distinguished least-element of that set, \emptyset , called the empty label.*
- *An associated binary operation, \bullet , on elements of \mathcal{L} we call concatenation.*

Finally, we require that \mathcal{L} be closed under concatenation: if ℓ_1 and ℓ_2 are in \mathcal{L} , then so is their concatenation $\ell_1 \bullet \ell_2$. \emptyset must act as a unit for concatenation.

Abstractly a label structure is a monoid (semigroup) over a poset. However, it would be straightforward to use any monoidal structure by defining $\emptyset \leq \ell$ for all $\ell \in \mathcal{L}$ and $\ell_1 \leq \ell_1 \bullet \ell_2$ and $\ell_2 \leq \ell_1 \bullet \ell_2$ for all $\ell_1, \ell_2 \in \mathcal{L}$.

An instantiation of a label structure that would be appropriate for a traditional effect type system is the free-monoid over the basis-set $\{\text{new}(l), \text{read}(l), \text{write}(l)\}$, parameterized by heap locations⁵ l . We then define the ordering as suggested above: $\emptyset \leq \ell$ for all labels ℓ , and $\ell_1 \leq \ell_1 \bullet \ell_2$ and $\ell_2 \leq \ell_1 \bullet \ell_2$ for all labels ℓ_1 and ℓ_2 . This can be roughly considered as ordering by inclusion. Here labels are traces of operations upon locations in the heap. This is why we use concatenation rather than union, \cup , as is done in some presentations — we do not want to imply a commitment that \gg concatenation \ll be commutative⁶. We may want the order in which effects are combined to matter: depending upon the precision necessary $\text{read}(l) \bullet \text{write}(l)$ and $\text{write}(l) \bullet \text{read}(l)$ need to be considered separate behaviors. It is certainly possible to admit a more conservative system.

⁵Perhaps more accurately regions?

⁶However, it is quite sensible to allow commutativity for certain subsequences. For example, given a label $\ell_1 \bullet \text{read}(l_1) \bullet \text{read}(l_2) \bullet \ell_2$ it is sound to commute the reads to be $\ell_1 \bullet \text{read}(l_2) \bullet \text{read}(l_1) \bullet \ell_2$.

Types	$\tau ::= \text{int} \mid \tau_1 \xrightarrow{\ell} \tau_2 \mid \dots$
Terms	$e ::= i \mid x \mid \lambda x : \tau . e \mid e_1 e_2 \mid \dots$
Term variable context	$\Gamma ::= \cdot \mid \Gamma, x : \tau$

Figure 2: The grammar for λ_{FX} .

$\frac{}{\Gamma \vdash_{\emptyset} i : \text{int}} \text{wft:int}$	$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\emptyset} x : \tau} \text{wft:var}$	$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash_{\emptyset} \lambda x : \tau . e : \tau_1 \xrightarrow{\ell} \tau_2} \text{wft:abs}$
$\frac{\Gamma \vdash_{\ell_1} e_1 : \tau_1 \xrightarrow{\ell} \tau_2 \quad \Gamma \vdash_{\ell_2} e_2 : \tau_1}{\Gamma \vdash_{\ell_1 \bullet \ell_2 \bullet \ell} e_1 e_2 : \tau_2} \text{wft:app}$		$\frac{\Gamma \vdash_{\ell_1} e : \tau_1 \quad \ell_1 \leq \ell_2 \quad \tau_1 \leq \tau_2}{\Gamma \vdash_{\ell_2} e : \tau_2} \text{wft:sub}$

Figure 3: The static semantics for λ_{FX} .

Another concrete instantiation of the label structure is a two level security lattice commonly used in information-flow type systems. We would take $\mathcal{L} = \{\perp, \top\}$, where $\perp \leq \top$ and $\ell \bullet \top = \top \bullet \ell = \top$ and \perp is the empty label and unit. Here \perp can be considered to correspond to low security information, and \top to high security information.

2.1 The λ_{FX} language

λ_{FX} is intended to be a core calculus that captures the essential features of effect type systems as originally developed by Gifford et. al [18]. The grammar for λ_{FX} can be found in Figure 2. The only difference from the simply-typed λ -calculus is that we annotate function types with a label. We write the type $\tau_1 \xrightarrow{\ell} \tau_2$ to indicate a function from values of type τ_1 to values of type τ_2 possibly producing some effect ℓ in the process.

We sketch the static semantics for λ_{FX} in Figure 3. We write $\Gamma \vdash_{\ell} e : \tau$ to mean »term e has type τ and may produce an effect ℓ with respect to the context Γ «. The key difference from the simply-typed λ -calculus is that the turnstiles of the judgements are annotated with a label. This label is intended as a conservative estimate of the effect that may be produced when evaluating the term. As we might expect, because abstractions suspend the evaluation of their bodies, the rule `wft:abs` captures the effect that may be produced by the abstraction body and records it in the function type.

It is also important to note that in λ_{FX} , values have no effects because they are computationally inert. Therefore values such as integers and functions have no intrinsic effect, and the conclusion of the typing rules `wft:int` and `wft:abs` are annotated with the empty label. This is also captured in the fact that term variables have no effect either, because in a call-by-value setting we will only ever substitute a value for a variable.

The rule for application, `wft:app`, strings together the effects in the conclusion to account for the order of evaluation. Finally, the typing rule `wft:sub` provides for subsumption of the effect

annotation and types. Because labels are ordered and occur in types, this induces a standard subtyping relationship.

2.1.1 Examples

We now use several different examples to illustrate how λ_{FX} could be used to model various kinds of effectful operations. Firstly, we consider extending λ_{FX} with reference cells. We declare a new syntactic category, l , of heap locations. In addition to a type $\text{ref}^l \tau$ we add three new term forms: allocation ($\text{ref}^l e$), dereferencing ($!e$), and assignment ($e_1 := e_2$). These terms are given the following static semantics.

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash_{l \bullet \text{new}(l)} \text{ref}^l e : \text{ref}^l \tau} \text{wft:ref} \qquad \frac{\Gamma \vdash_l e : \text{ref}^l \tau}{\Gamma \vdash_{l \bullet \text{read}(l)} !e : \tau} \text{wft:deref}$$

$$\frac{\Gamma \vdash_l e_1 : \text{ref}^l \tau \quad \Gamma \vdash_l e_2 : \tau}{\Gamma \vdash_{l \bullet \text{write}(l)} e_1 := e_2 : \text{ref}^l \tau} \text{wft:assn}$$

We also need to extend the language of effects with three primitive effects $\text{new}(l)$, $\text{read}(l)$, and $\text{write}(l)$. The term $\text{ref}^l e$ allocates a new storage cell at location l , initializing it with the result of evaluating e . This propagates an effect $\text{new}(l)$ indicating that a cell has been allocated at location l . Similarly the rules for dereferencing a cell and assigning to one propagate effects $\text{read}(l)$ and $\text{write}(l)$ respectively.

With these extensions, we can now write and type terms like the following

$$\lambda x : \text{int} . (\lambda y : \text{ref}^l \text{int} . 0)(\text{ref}^l x) : \text{int} \xrightarrow{\text{new}(l)} \text{int} \quad (1)$$

This function takes an integer argument to allocate and initialize a reference cell at location l . This allocation is not captured in the return type of the function, int , because the function does not actually return this reference cell. However, because the type system tracks effects, the allocation is captured in the overall type of the function, $\text{int} \xrightarrow{\text{new}(l)} \text{int}$. This says that every time this function is invoked it will cause a memory cell at location l to be allocated. This is just one example of how an effect system allows us to statically describe effectful behavior.

However, despite allowing us to track allocations, λ_{FX} is not powerful enough to soundly express deallocation. A naïve way that we might imagine trying to handle deallocation is to add a new effect $\text{free}(l)$ to indicate that the memory cell at location l has been deallocated. We would then augment the typing rule for dereferencing cells to check that the cell has not already been freed.

$$\frac{\Gamma \vdash_l e : \text{ref}^l \tau}{\Gamma \vdash_{l \bullet \text{free}(l)} \text{free } e : \text{int}} \text{wft:free} \qquad \frac{\Gamma \vdash_l e : \text{ref}^l \tau \quad \text{free}(l) \not\leq l}{\Gamma \vdash_{l \bullet \text{read}(l)} !e : \tau} \text{wft:deref-alt}$$

Unfortunately, this proves inadequate due to aliasing. Consider the following program

$$\begin{aligned} & (\lambda x : \text{ref}^l \text{int} . \\ & \quad (\lambda f : \text{int} \xrightarrow{\text{free}(l)} \text{int} . \lambda g : \text{int} \xrightarrow{\text{read}(l)} \text{int} . g(f0)) \\ & \quad (\lambda y : \text{int} . \text{free } x) : \text{int} \xrightarrow{\text{free}(l)} \text{int} \\ & \quad (\lambda z : \text{int} . !x) : \text{int} \xrightarrow{\text{read}(l)} \text{int})(\text{ref}^l 42) \end{aligned} \quad (2)$$

Here because the type system does not track the fact that the handle to the memory cell will become duplicated or aliased, it is possible to » accidentally « free the cell before reading from it (by invoking f before g). There are certainly other approaches, such a semantics where **free** removes $\text{new}(l)$ from the effect label and requires that the effect label contain a $\text{new}(l)$ effect before reading or writing. These alternative systems will all encounter similar difficulties with aliasing though.

While it may not possible to soundly handle direct deallocation in an effect system, by extending the language with regions [18] it is possible to model scoped allocation. One possible implementation of regions would be language construct for scoping heap locations

$$\frac{\Gamma \vdash e : \tau \quad l \notin \ell \quad l \notin \tau}{\Gamma \vdash \nu l . e : \tau} \text{wft:nu}$$

Here ν bind a new memory location l within the scope of e , but the two premises declare that no mention of l may escape through the effect label or the type. We assume that because Γ is given as an input, there it could not possibly reference l . Therefore, any allocations to l within e may be safely undone after it has finished evaluating. For example, consider this hypothetical swap function (assuming extensions for unit values and sequencing)

$$\begin{aligned} & \lambda x : \text{ref}^{l_1} \text{int} . \lambda y : \text{ref}^{l_2} \text{int} . \\ & \nu l . (\lambda z : \text{ref}^l \text{int} . x := (!y); y := (!z); \langle \rangle)(\text{ref}^l !x) \end{aligned} \quad (3)$$

Here a new reference cell is allocated to serve as a temporary store when swapping. However, because there are no references to l outside of the scope of the ν , as is enforced by the typing rule, the compiler may generate code to free the memory allocated at l .

Another example we might consider is the control effect of nontermination. One naïve implementation might be to simply add a fix-point operator to the language and a new primitive effect label diverge ⁷.

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash_{\bullet \text{diverge}} \mathbf{fix} \ x : \tau . e : \tau} \text{wft:fix}$$

It is not necessarily intuitive whether this is precise enough. Why not require that the body of fix-point carry a diverging effect?

$$\frac{\Gamma, x : \tau \vdash e : \tau \quad \text{diverge} \leq \ell}{\Gamma \vdash \mathbf{fix} \ x : \tau . e : \tau} \text{wft:fix-alt}$$

This alternate rule turns out to be inter-definable with wft:fix using subsumption. One other naïve idea would be to somehow attach a diverging effect to x . This actually illustrates a minor incompatibility of fix-points and λ_{FX} . We do not label variables, because in the core calculus of λ_{FX} the only binding construct is call-by-value abstraction and values have no effect. However, the fix-point is intrinsically call-by-name, so we will be substituting terms instead of values, which do have effects. So to obtain the best fit, we should have a construct for defining recursive functions – they will already be values when substituted.

$$\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2 \quad \text{diverge} \leq \ell}{\Gamma \vdash_{\emptyset} \mathbf{fix} \ f(x : \tau_1) . e : \tau_1 \xrightarrow{\ell} \tau_2} \text{wft:fix-fun}$$

⁷The subtlety of classifying effects can be emphasized by the fact that reference cells can be used to implement general recursion, but will not propagate a divergence effect.

The differences are subtle, but it does actually have a significant impact on the language.

Consider the these three implementations of a function that goes into an infinite loop. The first is typed using `wft:fix`, the second `wft:fix-alt`, and the last with `wft:fix-fun`.

$$f \triangleq \cdot \vdash_{\text{diverge}} (\mathbf{fix} \ f : \text{int} \xrightarrow{\emptyset} \text{int} . \lambda x : \text{int} . fx) : \text{int} \xrightarrow{\emptyset} \text{int} \quad (4)$$

$$f \triangleq \cdot \vdash_{\text{diverge}} (\mathbf{fix} \ f : \text{int} \xrightarrow{\emptyset} \text{int} . \lambda x : \text{int} . fx) : \text{int} \xrightarrow{\emptyset} \text{int} \quad (5)$$

$$f \triangleq \cdot \vdash_{\emptyset} \mathbf{fix} \ f(x : \text{int}) . fx : \text{int} \xrightarrow{\text{diverge}} \text{int} \quad (6)$$

The types of program Fragments 4 and 5 do not reveal that invoking them will lead to divergence. Instead the divergence is captured on the judgement's effect annotation. In the Fragment 6, however, it is correctly captured. This is important because of the following example

$$\cdot \vdash (\lambda g : \text{int} \xrightarrow{\text{diverge}} \text{int} . 3)f \quad (7)$$

If we used the first two semantics proposed for the fix-point operator, the above code fragment would appear to the type system as possibly diverging, despite the fact that `f` is simply thrown away.

Finally, we can also consider exceptions, which are notable as control effects that can be handled or masked. Here we introduce a new primitive effect label `exn(i)` indexed by some integer for distinguishing between them. The `throw` term can then be used to abandon the current control-flow context by raising the specified exception.

$$\frac{}{\Gamma \vdash_{\text{exn}(i)} \mathbf{throw} \ \text{exn}(i) : \tau} \text{wft:throw}$$

$$\frac{\Gamma \vdash_{\ell_1} e_1 : \tau \quad \Gamma \vdash_{\ell_2} e_2 : \tau \quad \text{exn}(i) \leq \ell_2 \quad \ell_2 \leq \ell_1 \bullet \text{exn}(i)}{\Gamma \vdash_{\ell_1} \mathbf{catch} \ \text{exn}(i) \ \mathbf{with} \ e_1 \ \mathbf{in} \ e_2 : \tau} \text{wft:catch}$$

Because `throw` never actually returns a value, we may give it any type we want, but it will be required to mark the term as causing an effect `exn(i)`. A `catch` term allows an exception `exn(i)` raised in `e2` to be handled by the code by code in `e1`. The precondition `exn(i) ≤ ℓ2` verifies that it is indeed possible for exception could be thrown (though there would be no harm in providing a handler regardless). The second precondition `ℓ2 ≤ ℓ1 • exn(i)` says that the effect annotation for the entire `catch` term, combined with the exception label, must be at least as large as the effect of the body. This means that `ℓ1` could be slightly smaller than `ℓ2`, because it need not include `exn(i)`, but it could be even larger if the exception handling code in `e1` causes effects of its own.

2.2 The λ_{SEC} language

We present λ_{SEC} as a canonical example of a language with a type system for tracking dependencies. It was originally developed by Zdancewic [48] as a core calculus for studying secure information-flow in programming languages. However, with minor modifications it would be a suitable basis for statically describing any number of dependency related analyses such as program slicing, call-tracking, or binding-time analysis. The grammar for λ_{SEC} can be found in Figure 4. Aside from the addition of a conditional, and some labels that were not found in λ_{FX} , it is essentially the same.

Types $\tau ::= \text{int}_\ell \mid \tau_1 \xrightarrow{\ell_1}_{\ell_2} \tau_2 \mid \dots$
 Terms $e ::= i \mid x \mid \lambda x : \tau . e \mid e_1 e_2 \mid \text{if0 } e_1 \text{ then } e_2 \text{ else } e_3 \mid \dots$
 Term variable context $\Gamma ::= \cdot \mid \Gamma, x : \tau$

Figure 4: The grammar for λ_{SEC} .

$$\frac{}{\Gamma \vdash_\ell i : \text{int}_\ell} \text{wft:int} \qquad \frac{x : \tau_1 \in \Gamma \quad \tau_1 \leq \tau_2 \ell \leq \tau_2}{\Gamma \vdash_\ell x : \tau_2} \text{wft:var} \qquad \frac{\Gamma, x : \tau_1 \vdash_{\ell_1} e : \tau_2}{\Gamma \vdash_{\ell_2} \lambda x : \tau . e : \tau_1 \xrightarrow{\ell_1}_{\ell_2} \tau_2} \text{wft:abs}$$

$$\frac{\Gamma \vdash_\ell e_1 : \tau_1 \xrightarrow{\ell_1}_{\ell_2} \tau_2 \quad \Gamma \vdash_\ell e_2 : \tau_1 \quad \ell \bullet \ell_2 \leq \ell_1}{\Gamma \vdash_\ell e_1 e_2 : \tau_2 \bullet \ell_2} \text{wft:app}$$

$$\frac{\Gamma \vdash_{\ell_1} e_1 : \text{int}_{\ell_2} \quad \Gamma \vdash_{\ell_1 \bullet \ell_2} e_2 : \tau \quad \Gamma \vdash_{\ell_1 \bullet \ell_2} e_3 : \tau}{\Gamma \vdash_{\ell_1} \text{if0 } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{wft:ifz}$$

$$\frac{\Gamma \vdash_{\ell_1} e : \tau_1 \quad \ell_2 \leq \ell_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash_{\ell_2} e : \tau_2} \text{wft:sub}$$

Figure 5: The static semantics for λ_{SEC} .

As we noted in Section 1.2, dependencies are tantalizing close to the dual of effects. For example, in λ_{SEC} , the type of all values is labelled. Comparing the static semantics of λ_{SEC} with λ_{FX} further emphasizes this connection. The typing rules for λ_{SEC} can be found in Figure 5. We write $\Gamma \vdash e : \tau$ to mean »assuming the dependencies ℓ , term e has type τ with respect to context Γ «. Much like λ_{FX} , the typing judgements of λ_{SEC} are annotated with a label. However, this label is not an output but an input. One can think of it as the dependency information of an imaginary »program counter« as it is about to begin execution of the term. Therefore, as is indicated in the rule `wft:int`, an integer constant will necessarily propagate those dependencies necessary to reach that control-flow point. We write the type of an integer as int_ℓ , meaning that it is an integer whose computation depended upon ℓ . Similarly, for variables, we must verify that any value substituted for a variable depends on at least as much as the current program counter. If not, we have rely on subsumption to raise the dependencies on the type of variable to be great enough. We write $\ell \leq \tau$ as shorthand for » ℓ is less than the label ℓ' on type τ «.

Again like λ_{FX} , the dependencies of the program counter are captured in `wft:abs` as part of the function type, $\tau_1 \xrightarrow{\ell_1} \tau_2$. Here ℓ_1 are the dependencies that the program counter must have before entering the function and ℓ_2 describes the dependencies that were required to create the function.

However, unlike λ_{FX} , the dependencies tracked by a function’s program counter are not propagated in the application rule `wft:app`. This is because it is merely intended as a precondition: it must be verified that the current program counter, plus any dependencies on the function itself, will have as many dependencies as the function where control is to be transferred. Furthermore, because the result of applying a function necessarily depends on what was needed to construct the function, we concatenate the result type with the label on the function. Again, $\tau \bullet \ell$ is shorthand for replacing the label ℓ' on type τ with $\ell' \bullet \ell$.

The astute reader will have noticed that none of the rules discussed so far actually alter the program counter. Therefore, for illustrative purposes we included conditionals in the language. Our conditional examines an integer and then chooses to execute one branch or another based upon whether it is zero. Whether the integer is zero or not is a function of its dependencies, ℓ_2 . Therefore, the decision to jump to one branch or another depends on ℓ_2 as well. Consequently, when control flow reaches one of the branches, the program counter will necessarily depend upon ℓ_2 as well. So when typing the branches we require that they be checked with a program counter $\ell_1 \bullet \ell_2$. It is edifying to note that a property of this type system is that for every well typed term $\Gamma \vdash e : \tau$ we have that $\ell \leq \tau$.

Finally, as with λ_{FX} , the ordering on labels induces a subsumption relationship for types, and it is convenient to include a subsumption rule. Not surprisingly, given the apparent duality, subsumption on the program counter must be contravariant. Why this is so is not entirely intuitive. However, consider if we allowed covariant subsumption.

$$\frac{\Gamma \vdash_{\ell_1} e : \tau_1 \quad \ell_1 \leq \ell_2 \quad \tau_1 \leq \tau_2}{\Gamma \vdash_{\ell_2} e : \tau_2} \text{wft:sub-alt}$$

Next consider a well-typed variable, $\cdot, x : \text{int}_\emptyset \vdash_\emptyset x : \text{int}_\emptyset$. The empty label is less than any other label, so $\emptyset \leq \ell$ for some ℓ . Furthermore, it is the case by reflexivity that $\text{int}_\emptyset \leq \text{int}_\emptyset$. Therefore using `wft:sub-alt` we could declare that $\cdot, x : \text{int}_\emptyset \vdash_\emptyset x : \text{xint}_\emptyset$. However, this clearly contradicts the precondition on `wft:var` – that the label on the type must be greater than the program counter⁸.

⁸There might be something in a version that allows the program counter to vary covariantly by requiring that $\ell_2 \leq \tau_2$.

2.2.1 Examples

Now we consider examples of concrete dependencies in λ_{SEC} . One of the most common instantiations of λ_{SEC} is as a security-type system. We might consider special integer constants that are high security and get a label of `secret`. For example, imagine a constant that is an integer representation of a password

$$\frac{\ell \leq \text{secret}}{\Gamma \vdash_{\tau} \text{password} : \text{int}_{\text{secret}}} \text{wft:pw}$$

It is then possible to trace through the parts of a program that will depend upon the high security information in `password`. In particular, we would be interested in knowing whether there is some part of the program that depends upon the value of `password` that should not. For example, consider the following test

$$\cdot \vdash_{\emptyset} \text{if0 password then 0 else 1} : \text{int}_{\text{secret}} \quad (8)$$

Just by looking at the type of this program fragment we know that it depends upon information that was high security.

However, for the purposes of comparison, it is more interesting look at extending λ_{SEC} with mutable references similar to those we added to λ_{FX} . Again hinting of a duality, the effects induced by allocation, reading, and writing are propagated via the label on the resulting value rather on the program counter.

$$\frac{\Gamma \vdash_{\tau} e : \tau}{\Gamma \vdash_{\tau} \text{ref}^l e : \text{ref}_{\ell \bullet \text{new}(l)}^l \tau} \text{wft:ref} \qquad \frac{\Gamma \vdash_{\tau_1} e : \text{ref}_{\ell_2}^l \tau}{\Gamma \vdash_{\tau_1} !e : \tau \bullet \ell_2 \bullet \text{read}(l)} \text{wft:deref}$$

$$\frac{\Gamma \vdash_{\tau_1} e_1 : \text{ref}_{\ell_2}^l \tau \quad \Gamma \vdash_{\tau_1} e_2 : \tau \quad \ell_2 \leq \tau}{\Gamma \vdash_{\tau_1} e_1 := e_2 : \text{ref}_{\ell_2 \bullet \text{write}(l)}^l \tau} \text{wft:assn}$$

For example, not only does an allocated reference cell accumulate dependencies from the program counter, but it also is combined with the primitive `new(l)`, indicating that it depended upon the allocation of a reference cell at location `l`. Similarly, the results of dereferencing and assignment are combined with labels `read(l)` and `write(l)` to indicate that the result depends upon having read or written to a heap location `l`. It is important to note that when dereferencing a memory cell, the result will also accumulate any dependencies that were necessary to create the cell.

Finally, the precondition $\ell_2 \leq \tau$ in `wft:assn` is actually quite confusing at first. We mentioned previously that for a well typed term $\Gamma \vdash_{\tau} e : \tau$ we have that $\ell \leq \tau$, so it seems like this property always be satisfied when a reference cell is created? The problem is that the label of a reference cell could be forced above its contents by a conditional test (among other things). Consider this example taken from Zdancewic [48]

$$\begin{aligned} & (\lambda x : \text{ref}_{\emptyset}^{l_1} \text{int} . \lambda y : \text{ref}_{\emptyset}^{l_2} \text{int} \\ & \quad \text{let } z = (\text{if0 password then } x \text{ else } y) \text{ in} \\ & \quad \quad z := 1 \\ & \quad \dots)(\text{ref}^{l_1} 0)(\text{ref}^{l_2} 0) \end{aligned} \quad (9)$$

Inside the conditional the dependency of the program counter has been raised to **secret** because of the test of password. This means the label on z must at least depend upon **secret**. However, if we were able to go ahead and write 1 to z , we could then later read back both the contents of x and y to determine whether password is zero and the dependency will be lost. Therefore, we must restrict writes so that the label on the reference cell is less than its contents to ensure that the dependency is captured.

It is illustrative to contrast the typing for the simple example of memory allocation we presented for λ_{FX} with its equivalent in λ_{SEC}

$$\cdot \vdash_{\emptyset} \lambda x : \text{int}_{\emptyset} . (\lambda y : \text{ref}_{\text{new}(l)}^1 \text{int}_{\emptyset} . 0)(\text{ref}^1 x) : \text{int}_{\emptyset} \xrightarrow{\emptyset} \text{int}_{\emptyset} \quad (10)$$

Here, because creating the function and its result have no dependency upon the memory cell allocated, the allocation is not reflected in the types at all.

Like λ_{FX} , because of aliasing it is not possible to soundly model explicit deallocation in λ_{SEC} . It does not seem that the use of a region system for scoped allocation has been considered in the context of a dependency system.

We can also examine nontermination in a dependency type system. Interestingly, treating nontermination as a dependency does seem to offer some advantages. We might consider adding a fix-point operator in the following manner

$$\frac{\Gamma, x : \tau \vdash e : \tau \quad \text{diverge} \leq \tau}{\Gamma \vdash \mathbf{fix} \ x : \tau . e : \tau} \text{wft:fix}$$

This rule for fix-points looks very similar to one we considered for λ_{FX} . Firstly, because both values and terms carry their dependency information in their type, the call-by-name nature of the fix-point does not pose a problem. Once again, as the program counter is an input, we instead check that the overall type of the fixpoint has a label that is greater than that of **diverge**. One particular advantage of this is that any uses of x inside the body of the fix-point will correctly indicate that the result could depend upon a nonterminating computation.

Revisiting our earlier example of an infinite loop

$$\vdash_{\emptyset} (\mathbf{fix} \ f : \text{int}_{\emptyset} \xrightarrow{\emptyset} \text{diverge} \ \text{int}_{\text{diverge}} . \lambda x : \text{int}_{\perp} . fx) : \text{int}_{\emptyset} \xrightarrow{\emptyset} \text{diverge} \ \text{int}_{\text{diverge}} \quad (11)$$

we obtain a much more precise type. Not only does the type of function indicate that the entire expression is potentially diverging to begin with, it indicates that the integer produced by the function could be dependent upon a diverging computation as well.

Given our intuitions about duality, extending λ_{SEC} with exceptions and exception handling is very similar to λ_{FX} , but like our other examples, we push the dependency information out through the type

$$\frac{\ell \bullet \text{exn}(i) \leq \tau}{\Gamma \vdash \mathbf{throw} \ \text{exn}(i) : \tau} \text{wft:throw}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \text{exn}(i) \leq \tau_2 \quad \tau_2 \leq \tau_1 \bullet \text{exn}(i)}{\Gamma \vdash \mathbf{catch} \ \text{exn}(i) \ \mathbf{with} \ e_1 \ \mathbf{in} \ e_2 : \tau_1} \text{wft:catch}$$

The rule for **throw** captures the fact that, as before, we can give this term any type we want, because it never returns to the current context, but that it must capture the program counter dependencies and the exception dependency. Again the first precondition in `wft:catch` verifies that the exception could be thrown, and the second allows weakening the result type to exclude the exception dependency.

2.3 The λ_{\circ} language

λ_{FX} was presented as an example of a type system for capturing effectful computation. λ_{SEC} was presented as a prototypical type system for representing dependency relationships in programs. Interestingly, despite the apparently dual relationship between effects and dependencies, we can embed them both into a single language, λ_{\circ} .

We call \circ the »lax« modality because λ_{\circ} is the computational interpretation of what is known as lax logic [15]. Lax logic is in turn actually the logical interpretation of Eugenio Moggi’s monadic meta-language [31], which he developed for reasoning about programs with effectful behavior. In fact, the strong monadic structure of the lax modality has become so important to the handling of effects in purely functional languages, such as Haskell [36], that it is often simply called a »monad«. Which, as we will see, is actually a bit of a misnomer.

2.3.1 Category theoretic foundations

Before attempting to explain λ_{\circ} itself, we will take a detour into the category theoretic foundations of monads. We assume the reader has a cursory understanding of category theory. The uninitiated reader may wish to consult an introductory text [26, 39, 6].

Definition 2.2 *Monads, also known as triples, are categorical structures consisting of three components:*

- A functor, $F : \mathcal{C} \rightarrow \mathcal{C}$, from a category \mathcal{C} to itself. In other words, F is an endofunctor.
- A natural transformation, $\eta : \text{Id}_{\mathcal{C}} \rightarrow F$, from the identity functor on \mathcal{C} to F .
- A second natural transformation, $\mu : F \circ F \rightarrow F$, from the composition of F with itself to F .

Additionally, a monad $\langle F, \eta, \mu \rangle$ must satisfy the following commutative diagrams

$$\begin{array}{ccc}
 F \circ F \circ F & \xrightarrow{F \circ \mu} & F \circ F \\
 \mu \circ F \downarrow & & \downarrow \mu \\
 F \circ F & \xrightarrow{\mu} & F
 \end{array}
 \qquad
 \begin{array}{ccccc}
 & & F \circ \eta & & \eta \circ F \\
 & & \downarrow & & \downarrow \\
 F & \xrightarrow{\eta} & F \circ F & \xleftarrow{\eta \circ F} & F \\
 & \searrow \text{Id}_F & \downarrow \mu & \swarrow \text{Id}_F & \\
 & & F & &
 \end{array}$$

where Id_F is the identity functor F .

The natural transformation η is called the *unit* of the monad, and μ is sometimes called the *join*. One way of understanding monads is as a categorical generalization of the algebraic concept of monoids. The first diagram above can be understood as saying that the join operator is associative, like a monoidal operator. The second diagram specifies that the unit operator is an algebraic unit of the join operator. In fact, it is possible to describe any given monoid as a monad over the category of sets.

Despite all the fuss about monads, they are not really what is used in practice⁹. Even in Moggi's original paper on the monadic meta-language [31], Kleisli triples are used instead. There exists a one-to-one correspondence between monads and Kleisli triples, so this conflation is not entirely unjustified, but it is confusing. Regardless, there are subtle differences.

Definition 2.3 A Kleisli triple is a categorial structure also consisting of three components:

- A functor, $F : \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C})$, a functor from objects of a category \mathcal{C} to themselves.
- A family of morphisms $\eta_X : X \rightarrow F(X)$, for all $X \in \text{Obj}(\mathcal{C})$.
- An operator \star such that for any morphism $f : X \rightarrow F(Y)$ in \mathcal{C} , where $X, Y \in \text{Obj}(\mathcal{C})$, then $f^\star : F(X) \rightarrow F(Y)$.¹⁰

Furthermore, a Kleisli triple must satisfy the following equational properties:

- $\eta_X^\star = \text{id}_{F(X)}$.
- $f^\star \circ \eta_X = f$ for $f : X \rightarrow F(Y)$.
- $g^\star \circ f^\star = (g^\star \circ f)^\star$ for $f : X \rightarrow F(Y)$ and $g : Y \rightarrow F(Z)$.

As with monads, η_X are called units. The operator \star is often called the *extension*. It is often convenient to treat the extension as a binary operator, called *bind*, from an objects $F(X) \in \mathcal{C}$ and morphisms $f : X \rightarrow F(Y)$ in \mathcal{C} to morphisms $f^\star : F(X) \rightarrow F(Y)$. In practical programming the bind operator turns out to be a much more intuitive way of working with extensions and sequence computations. So for the remainder of our presentation we will focus upon Kleisli triples with a bind operator.

Returning to programming languages, we can consider the category where objects are types and morphisms $f : \tau_1 \rightarrow \tau_2$ are terms of type τ_2 with a free variable of type τ_1 . In this context, it straightforward to show that any number of type constructors and associated operations form a Kleisli triple. For example, the list type constructor could be used as the functor, along with the function to construct a singleton list as the unit and map composed with flatten as the bind operator

$$\begin{aligned} F &\triangleq \text{list} \\ \eta_\tau &\triangleq \lambda x : \tau . [x] \\ \star &\triangleq \lambda x : F(\tau_1) . \lambda f : \tau_1 \rightarrow F(\tau_2) . \text{flatten}(\text{map } f \ x) \end{aligned}$$

However, as Moggi suggested, it is more useful to think abstractly of objects τ as being values of τ and objects $F(\tau)$ as computations of τ . This led Moggi to call the functor F a *notion of computation*, because it abstracts away from the true values a computation may produce. For example, the following effectful computations may be conveniently abstracted in with the following definitions:

- **state:** $F \triangleq \lambda \alpha . \sigma \rightarrow (\alpha \times \sigma)$, where σ is the type of the state of the system.
- **exceptions:** $F \triangleq \lambda \alpha . \alpha + \sigma$, where σ is the type of an exception.

⁹Furthermore, we elide the requirement that monads be »strong«, as it is unlikely to add anything to this presentation.

¹⁰I think it would be more precise to describe \star in terms of categorical exponentials, but no one seems to do this, and it would require cluttering the presentation with uses of `eval` and `curry`. Of course, it would also require that \mathcal{C} be Cartesian.

Types	$\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \text{O}\tau \mid \dots$
Terms	$e ::= i \mid x \mid \lambda x : \tau_1 . e \mid e_1 e_2 \mid \mathbf{val} E \mid \dots$
Expressions	$E ::= [e] \mid \mathbf{let val} x = e \mathbf{in} E \mid \dots$
Term variable context	$\Gamma ::= \cdot \mid \Gamma, x : \tau$

Figure 6: The grammar for λ_{O} .

$\frac{}{\Gamma \vdash i : \text{int}} \text{wft:int}$	$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{wft:var}$	$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau . e : \tau_1 \rightarrow \tau_2} \text{wft:abs}$
$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{wft:app}$		$\frac{\Gamma \vdash e \div \tau}{\Gamma \vdash \mathbf{val} e : \text{O}\tau} \text{wft:val}$
$\frac{\Gamma \vdash e : \text{O}\tau_1 \quad \Gamma, x : \tau_1 \vdash E \div \tau_2}{\Gamma \vdash \mathbf{let val} x = e \mathbf{in} E \div \tau_2} \text{wfe:let}$		
$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] \div \tau} \text{wfe:unit}$		

Figure 7: The static semantics for λ_{O} .

2.3.2 A judgemental reconstruction

If we wish to abstract away from specific instantiations of a notion of computation, λ_{O} provides idealized language for distinguishing between values and computations. The grammar is given in Figure 6. Our presentation follows the judgemental precepts of Pfenning and Davies [38], and separates the language into »pure« terms and »effectful« expressions. We chose to use this definition because we felt that it made the operational semantics less ad-hoc. We take complete programs to be expressions rather than terms.

The modal type constructor O is used as a generic monadic functor. The term $\mathbf{val} E$ is an injection from expressions into the language of terms. We use the square bracket operator $[\cdot]$ as the unit operator for lifting terms to expressions; $\mathbf{let val} x = e \mathbf{in} E$ is the bind operator that essentially sequences computations – the computations in e are forced to occur before those in E . This is reflected in the static semantics as presented in Figure 7. We write the judgement $\Gamma \vdash E \div \tau$ to mean »expression E has type τ with respect to context Γ «.

It important to note that \mathbf{val} acts as a suspension or thunk, with bind forcing their evaluation. Because they are not entirely obvious, Figure 8 shows some selected reduction rules for λ_{O} .

2.3.3 Examples

It is straightforward to extend λ_{O} to capture specific notions of effectful computation by expanding the language of expressions. For example, the static semantics of our running example of reference

$$\begin{array}{c}
\frac{e \rightsquigarrow e'}{[e] \rightsquigarrow [e']} \text{ ev:unit} \qquad \frac{e \rightsquigarrow e'}{\mathbf{let\ val\ } x = e \mathbf{ in\ } E \rightsquigarrow \mathbf{let\ val\ } x = e' \mathbf{ in\ } E} \text{ ev:letv1} \\
\\
\frac{E_1 \rightsquigarrow E'_1}{\mathbf{let\ val\ } x = \mathbf{val\ } E_1 \mathbf{ in\ } E_2 \rightsquigarrow \mathbf{let\ val\ } x = \mathbf{val\ } E'_1 \mathbf{ in\ } E_2} \text{ ev:letv2} \\
\\
\frac{}{\mathbf{let\ val\ } x = \mathbf{val\ } [v] \mathbf{ in\ } E \rightsquigarrow E[v/x]} \text{ ev:letv3}
\end{array}$$

Figure 8: Selected reduction rules for $\lambda_{\mathcal{O}}$.

cells would be given as

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}^l e \div \mathbf{ref}^l \tau} \text{ wfe:ref} \qquad \frac{\Gamma \vdash e : \mathbf{ref}^l \tau}{\Gamma \vdash !e \div \tau} \text{ wfe:deref} \qquad \frac{\Gamma \vdash e_1 : \mathbf{ref}^l \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 \div \mathbf{ref}^l \tau} \text{ wfe:assn}$$

The key idea behind the semantics is that because each of these operations produce some kind of effect, we place them in the language of expressions. Therefore, the monadic structure of the lax modality ensure that the bind operator must be used to force and sequence effects.

The operational semantics for our monadic reference cells are again not entirely obvious, so we present the critical reductions below

$$\begin{array}{c}
\frac{}{\langle H, \mathbf{ref}^l v \rangle \rightsquigarrow \langle H \cup \{l \mapsto v\}, [l] \rangle} \text{ ev:ref} \qquad \frac{}{\langle H, !l \rangle \rightsquigarrow \langle H, [H(l)] \rangle} \text{ ev:deref} \\
\\
\frac{}{\langle H, l := v \rangle \rightsquigarrow \langle H \cup \{l \mapsto v\}, [l] \rangle} \text{ ev:assn}
\end{array}$$

We have assumed that all reduction rules have been augmented with an accompanying heap. Essentially, the result of each of these reductions is a value, but it is lifted to be a computation with the unit operator.

Our simple running example of memory allocation naïvely becomes

$$\cdot \vdash \lambda x : \text{int} . (\lambda y : \mathcal{O}\mathbf{ref}^l \text{int} . 0)(\mathbf{val\ ref}^l x) : \text{int} \rightarrow \text{int} \tag{12}$$

Notice that because the reference cell that we allocated is never used, no bind is necessary. Furthermore, there is no indication in the type that any effect occurred. This may seem a little strange, until realizing that this function does not actually implement the same behavior as the similar looking functions we defined for λ_{FX} and λ_{SEC} . This is because **val** suspends the computation of its body, and no allocation ever actually takes place. A semantically equivalent version would look like

$$\cdot \vdash \lambda x : \text{int} . \mathbf{val\ (let\ val\ } z = (\mathbf{val\ ref}^l x) \mathbf{ in\ } [(\lambda y : \mathbf{ref}^l \text{int} . 0)z]) : \text{int} \rightarrow \mathcal{O}\text{int} \tag{13}$$

This revised version correctly captures the behavior, and now the type does reflect that the result will live in the space of computations because of the allocation that occurs.

Interestingly, the sequencing of the monad can be used to track dependencies too. Consider our example from secure information-flow, where we want to keep track of which parts of a program may depend upon high security information. We can again introduce a high security constant

$$\frac{}{\Gamma \vdash \text{password} \div \text{int}} \text{wfe:pw}$$

Here instead of using a high security label, we simply place `password` into the language of expressions. For any piece of code to actually make use of `password`, it must first bind it, as such

$$\cdot \vdash \text{let val } x = \text{password in } \dots \div \tau \quad (14)$$

However, because the bind operation requires that the body must be an expression as well, the result of the entire expression will be forced to live in the world of computations as well. Therefore, if we were interested in a simple two level security lattice, we would be able to distinguish those parts of the program that depend upon high security information by the fact that they have monadic types.

Returning to effects, nontermination in λ_{O} is also captured by making the fix-point operator an expression

$$\frac{\Gamma, x : \text{O}\tau \vdash E \div \tau}{\Gamma \vdash \text{fix } x : \tau . E \div \tau} \text{wfe:fix}$$

Any potentially diverging computation must be explicitly forced by using `bind`. Again because the operational semantics is not immediately apparent, we present the reduction rule for the fix-point operator

$$\frac{}{\text{fix } x : \tau . E \rightsquigarrow E[\text{val } (\text{fix } x : \tau . E)/x]} \text{ev:fix}$$

Our running example of an infinite loop would become

$$\cdot \vdash \text{fix } f : \text{int} \rightarrow \text{int} . \text{let val } g = f \text{ in } [(\lambda x : \text{int} . gx)] \div \text{int} \rightarrow \text{int} \quad (15)$$

Exceptions are relatively straightforward extension as well.

$$\frac{}{\Gamma \vdash \text{throw } \text{exn}(i) \div \tau} \text{wfe:throw} \qquad \frac{\Gamma \vdash e : \text{O}\tau \quad \Gamma \vdash E \div \tau}{\Gamma \vdash \text{catch } \text{exn}(i) \text{ with } E \text{ in } e \div \tau} \text{wfe:catch}$$

As we would expect, `throw` must be an expression because it causes an effect. In this particular presentation, `catch` must actually duplicate some of the functionality of the bind operator, forcing the computation of the body `e` to check whether an exception occurred and then injecting the result back into the monad. However, we might imagine making `catch` a term, and having it behave a little more like the versions in λ_{FX} and λ_{SEC}

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash E \div \tau}{\Gamma \vdash \text{catch } \text{exn}(i) \text{ with } e \text{ in } E : \tau} \text{wft:catch-alt}$$

Types $\tau ::= \dots \mid \mathbb{O}_{(\ell_1, \ell_2)}\tau \mid \dots$

Figure 9: Extending $\lambda_{\mathbb{O}}$ with labels.

$$\begin{array}{c}
\frac{\Gamma \vdash E \dot{\div}_{(\ell_1, \ell_2)} \tau}{\Gamma \vdash \mathbf{val} E : \mathbb{O}_{(\ell_1, \ell_2)}\tau} \text{wft:vall} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] \dot{\div}_{(\emptyset, \top)} \tau} \text{wfe:unitl} \\
\frac{\Gamma \vdash e_1 : \mathbb{O}_{(\ell_1, \ell'_1)}\tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \dot{\div}_{(\ell_2, \ell'_2)} \tau_2 \quad \ell_1 \leq \ell_2 \quad \ell'_2 \leq \ell'_1}{\Gamma \vdash \mathbf{let val} x = e_1 \mathbf{in} e_2 \dot{\div}_{(\ell_2, \ell'_2)} \tau_2} \text{wfe:letl}
\end{array}$$

Figure 10: Labeled semantics for $\lambda_{\mathbb{O}}$.

This version of **catch** ostensibly allows us to escape the monad by handling the exception. However, this rule is actually unsound. It does not account for the fact that we have been placing other effects into the monad that **catch** will not handle. Furthermore, even if we limited effects to exceptions, as long as there is more than one distinct exception, we cannot be sure we handled the only one that might have been thrown. This is just one motivation for considering an extension of $\lambda_{\mathbb{O}}$ where the lax modality is labelled.

2.3.4 Labelled $\lambda_{\mathbb{O}}$

While we have seen that Kleisli triples provide a clean and structured approach to modeling some effectful computations, we have also found that the type system of $\lambda_{\mathbb{O}}$ is not sufficiently descriptive. A term with type $\mathbb{O}\tau$ is some computation producing a value τ , but the nature of the computation is completely hidden. Therefore, it is useful to consider extending $\lambda_{\mathbb{O}}$ so that it has not just one monadic modality, but an entire lattice of them. This approach has been taken by both Wadler [45] and Abadi et. al. [1] for effects and dependencies respectively, but here we chose to use a version derived from Cray, Kliger, and Pfenning [12]. In Figures 9 and 7 we give the grammar and semantics for such an extension.

Particularly notable is that we give the lax modality, not one, but two labels. The first label acts as a lower-bound on the effects of a computation, while the second label serves as an upper-bound. Accordingly, the first label must always be less in the order than the second, and they vary co- and contra-variantly with respect to subsumption. Because the second label varies contra-variantly it is convenient to assume that in addition to the distinguished empty label that we have a distinguished top label \top that is larger than all other labels.

The new typing rule **wfe:unit** for the unit reflects the fact that a value itself is computationally inert, and therefore it can be typed with the smallest and largest labels respectively. The typing rule **wfe:letl** can be seen as capturing both the essence of an effect system and a dependency system.

It is straightforward to extend $\lambda_{\mathbb{O}}$ to capture specific notions of effectful computation by ex-

$$\begin{aligned}
\llbracket \text{int} \rrbracket &\triangleq \text{int} \\
\llbracket \tau_1 \xrightarrow{\ell} \tau_2 \rrbracket &\triangleq \llbracket \tau_1 \rrbracket \rightarrow \mathcal{O}_{(\ell, \top)} \llbracket \tau_2 \rrbracket \\
\llbracket i \rrbracket &\triangleq [i] \\
\llbracket x \rrbracket &\triangleq [x] \\
\llbracket \lambda x : \tau . e \rrbracket &\triangleq [\lambda x : \llbracket \tau \rrbracket . \mathbf{val} \llbracket e \rrbracket] \\
\llbracket e_1 e_2 \rrbracket &\triangleq \mathbf{let val } x_1 = (\mathbf{val} \llbracket e_1 \rrbracket) \mathbf{ in let val } x_2 = (\mathbf{val} \llbracket e_2 \rrbracket) \mathbf{ in } [x_1 x_2] \\
\llbracket \cdot \rrbracket &\triangleq \cdot \\
\llbracket \Gamma, x : \tau \rrbracket &\triangleq \llbracket \Gamma \rrbracket, x : \llbracket \tau \rrbracket \\
\llbracket \Gamma \vdash_{\tau} e : \tau \rrbracket &\triangleq \llbracket \Gamma \rrbracket \vdash \llbracket e \rrbracket \div_{(\ell, \top)} \llbracket \tau \rrbracket
\end{aligned}$$

Figure 11: Translation of λ_{FX} into λ_{O}

panding the language of expressions. For example, the static semantics of our running example of reference cells would be given as:

$$\begin{array}{c}
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}^l e \div_{(\text{new}(l, \top))} \mathbf{ref}^l \tau} \text{wfe:refl} \qquad \frac{\Gamma \vdash e : \mathbf{ref}^l \tau}{\Gamma \vdash !e \div_{(\text{read}(l, \top))} \tau} \text{wfe:deref} \\
\frac{\Gamma \vdash e_1 : \mathbf{ref}^l \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 \div_{(\text{write}(l, \top))} \mathbf{ref}^l \tau} \text{wfe:assnl}
\end{array}$$

2.3.5 Encoding λ_{FX} and λ_{SEC} into λ_{O}

As alluded to earlier, it actually possible to encode programs written in λ_{FX} and λ_{SEC} into λ_{O} (when extended with labels). Figures 11 and 12 give the translations of λ_{FX} and λ_{SEC} , respectively, into λ_{O} . The translation for λ_{FX} is based upon Wadler’s [45] and the translation for λ_{SEC} is derived from Crary, Klinger, and Pfenning [12].

The translations are strikingly reminiscent of continuation passing style conversion [29]. In fact, that is an interesting relationship between continuations and monads; Filinski has shown how to implement monads in terms of continuations [16] and continuation passing style can be easily expressed as an instance of monad. Exploring this relationship further is beyond the scope of our present survey, however. The idea is essentially the same, however. λ_{O} requires that computational effects and dependencies be sequenced through a series of bind operations, just as continuation passing style makes evaluation order explicit by sequencing all computation with a continuation.

2.4 The language $\lambda_{\square\Diamond}$

In Section 2.3, we described the \mathcal{O} type constructor as a modality. This terminology comes from logic. The two classic examples of modalities from logic are *necessity* and *possibility*. With respect

$$\begin{aligned}
\llbracket \text{int}_\ell \rrbracket &\triangleq \mathcal{O}_{(\ell, \top)} \text{int} \\
\llbracket \tau_1 \xrightarrow{\ell_1}_{\ell_2} \tau_2 \rrbracket &\triangleq \mathcal{O}_{(\ell_2, \top)} (\llbracket \tau_1 \rrbracket \rightarrow \mathcal{O}_{(\emptyset, \ell_1)} \llbracket \tau_2 \rrbracket) \\
\llbracket i \rrbracket &\triangleq [i] \\
\llbracket x \rrbracket &\triangleq [x] \\
\llbracket \lambda x : \tau . e \rrbracket &\triangleq [\lambda x : [\tau] . \mathbf{val} \llbracket e \rrbracket] \\
\llbracket e_1 e_2 \rrbracket &\triangleq \mathbf{let val } x_1 = (\mathbf{val} \llbracket e_1 \rrbracket) \mathbf{ in let val } x_2 = (\mathbf{val} \llbracket e_2 \rrbracket) \mathbf{ in (let val } x = [x_1 x_2] \mathbf{ in } [x]) \\
\llbracket \cdot \rrbracket &\triangleq \cdot \\
\llbracket \Gamma, x : \tau \rrbracket &\triangleq [\Gamma], x : [\tau] \\
\llbracket \Gamma \vdash_\ell e : \tau \rrbracket &\triangleq [\Gamma] \vdash \llbracket e \rrbracket \div_{(\emptyset, \ell)} [\tau]
\end{aligned}$$

Figure 12: Translation of λ_{SEC} into $\lambda_{\mathcal{O}}$

to a Kripke possible world semantics [24], truth is relative to a particular world. Generally the »universe« is specified as a set of worlds and a relation, \prec , on these worlds. A statement τ is then said to be necessarily true, $\Box\tau$, if τ is a truth in *all* worlds accessible via \prec to the present one. Conversely, a statement τ is said to be possibly true, $\Diamond\tau$ if it is a truth in *some* world accessible via \prec to the present one.

How does this thread weave back into our story? In Pfenning and Davies' judgemental reconstruction of modal necessity and possibility [38], they were able to show that the lax modality can be cleanly decomposed into a combination of possibility and necessity. A variation on their encoding is shown in Figure 13. The fact that the lax modality seems to have properties of both necessity and possibility had already been noted by Fairtlough and Mendler [15]. This naturally leads to the question of whether possibility and necessity in some way provide a more fundamental treatment of effects and dependencies than the lax modality.

Interestingly, \Box forms a comonad (or more accurately a co-Kleisli triple) and \Diamond a monad. We expect the reader can dualize the definition of a monad without much difficulty, but because Kleisli triples were not presented diagrammatically we will for convenience present the definition of a co-Kleisli tripe here.

Definition 2.4 *A co-Kleisli triple is a categorical structure consisting of three components:*

- *A functor, $F : \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C})$, a functor from objects of a category \mathcal{C} to themselves.*
- *A family of morphisms $e_X : F(X) \rightarrow X$, for all $X \in \text{Obj}(\mathcal{C})$.*
- *An operator $*$ such that for any morphism $f : F(X) \rightarrow Y$ in \mathcal{C} , where $X, Y \in \text{Obj}(\mathcal{C})$, then $f^* : F(X) \rightarrow F(Y)$.*

Furthermore, a co-Kleisli triple must satisfy the following equational properties:

- $e_X^* = \text{id}_{F(X)}$.

$$\begin{aligned}
\llbracket \text{int} \rrbracket &\triangleq \text{int} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket &\triangleq \square \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket \\
\llbracket \circ \tau \rrbracket &\triangleq \diamond \square \llbracket \tau \rrbracket \\
\\
\llbracket i \rrbracket &\triangleq i \\
\llbracket x \rrbracket &\triangleq x \\
\llbracket \lambda x : \tau . e \rrbracket &\triangleq \lambda x : \square \llbracket \tau \rrbracket . \mathbf{let\ box\ } x = x \mathbf{ in } \llbracket e \rrbracket \\
\llbracket e_1 e_2 \rrbracket &\triangleq \llbracket e_1 \rrbracket (\mathbf{box} \llbracket e_2 \rrbracket) \\
\llbracket \mathbf{val\ } E \rrbracket &\triangleq \mathbf{dia} \llbracket E \rrbracket \\
\\
\llbracket \llbracket e \rrbracket \rrbracket &\triangleq \mathbf{box} \llbracket e \rrbracket \\
\llbracket \mathbf{let\ val\ } x = e \mathbf{ in } E \rrbracket &\triangleq \mathbf{let\ dia\ } x = \llbracket e \rrbracket \mathbf{ in } (\mathbf{let\ dia\ } y = (\mathbf{let\ box\ } x = x \mathbf{ in } \mathbf{dia} \llbracket E \rrbracket)) \mathbf{ in } \llbracket y \rrbracket
\end{aligned}$$

Figure 13: Unlabeled translation of λ_{\circ} into $\lambda_{\square\diamond}$

Types	$\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \square_{\ell} \tau \mid \diamond_{\ell} \tau \mid \dots$
Terms	$e ::= i \mid x \mid \lambda x : \tau . e \mid e_1 e_2 \mid \mathbf{box\ } e \mid \mathbf{let\ box\ } x = e_1 \mathbf{ in } e_2 \mid \mathbf{dia\ } E \mid \dots$
Expressions	$E ::= [e] \mid \mathbf{let\ dia\ } x = e \mathbf{ in } E \mid \dots$
Term variable context	$\Gamma ::= \cdot \mid \Gamma, x : \tau[\ell]$

Figure 14: The grammar for $\lambda_{\square\diamond}$.

- $\epsilon_Y \circ f^* = f$ for $f : F(X) \rightarrow Y$.
- $f^* \circ g^* = (f \circ g^*)^*$ for $f : F(X) \rightarrow Y$ and $g : F(Y) \rightarrow Z$.

Here we call ϵ the *co-unit* of the co-triple, and $*$ the *co-extension*. As with Kleisli triples, it is often more convenient to consider $*$ to be a binary operator called the *co-bind*.

It has actually been suggested previously that a comonadic structure might be more appropriate for representing effects that arise from the context in which a program fragment may execute [23, 33, 32, 34]. Kieburtz was the first to propose the use of comonads, and recently Nanevski took these ideas further in a nominal account of effects in modal type systems. Nanevski's calculus builds on the idea of fresh names developed by Gabbay and Pitts [17] and the judgemental account of modal logic by Pfenning and Davies. Nanevski's thesis is that modal necessity provides a way to demarcate which bits of code are impure and enforce the correct propagation of effects and that modal possibility handles the single-threading of effects and globalizing their scope [32].

The essential use of nominality in Nanevski's calculus seem to be delimiting the scope of specific effects, which seems to have a close correspondence to regions, though he does not note any connection. For our purposes, his use of names seems to be completely definable in terms of our labels, so for uniformity we choose to use labels in our presentation. The grammar for $\lambda_{\square\diamond}$ can be found in Figure 14.

$$\begin{array}{c}
\frac{}{\Gamma \vdash i : \text{int}[\ell]} \text{wft:int} \qquad \frac{x : \tau[\ell_1] \in \Gamma \quad \ell_1 \leq \ell_2}{\Gamma \vdash x : \tau[\ell_2]} \text{wft:var} \qquad \frac{\Gamma, x : \tau_1[\emptyset] \vdash e : \tau_2[\emptyset]}{\Gamma \vdash \lambda x : \tau . e : \tau_1 \rightarrow \tau_2[\ell]} \text{wft:abs} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2[\ell] \quad \Gamma \vdash e_2 : \tau_1[\ell]}{\Gamma \vdash e_1 e_2 : \tau_2[\ell]} \text{wft:app} \qquad \frac{\Gamma \vdash e : \tau[\ell_1]}{\Gamma \vdash \mathbf{box} e : \square_{\ell_1} \tau[\ell_2]} \text{wft:box} \\
\\
\frac{\Gamma \vdash e_1 : \square_{\ell_1} \tau_1[\ell_2] \quad \Gamma, x : \tau_1[\ell_1] \vdash e_2 : \tau_2[\ell_2]}{\Gamma \vdash \mathbf{let box} x = e_1 \mathbf{in} e_2 : \tau_2[\ell_2]} \text{wft:letb} \qquad \frac{\Gamma \vdash E \div \tau[\ell_1]}{\Gamma \vdash \mathbf{dia} E : \diamond_{\ell_1} \tau[\ell_2]} \text{wft:dia} \\
\\
\frac{\Gamma \vdash e : \tau_2[\ell]}{\Gamma \vdash [e] \div \tau_2[\ell]} \text{wfe:unit} \qquad \frac{\Gamma \vdash e : \diamond_{\ell_1} \tau_1[\ell_2] \quad \Gamma, x : \tau[\ell_1] \vdash E \div \tau_2[\ell_2]}{\Gamma \vdash \mathbf{let dia} x = e \mathbf{in} E \div \tau_2[\ell_2]} \text{wfe:letd}
\end{array}$$

Figure 15: The static semantics for $\lambda_{\square\Diamond}$.

Superficially, the grammar of $\lambda_{\square\Diamond}$ is quite similar to λ_{\square} with labels. Both languages divide programs into terms and expressions. Instead of **val** we have **dia** for injecting expressions into the term language. The unit operator $[\cdot]$ and **let dia** $x = e$ **in** E correspond to the monadic unit and bind. The terms **box** and **let box** roughly correspond to co-bind and the co-unit of a comonad, but this correspondence is a little less clear than the unit and bind of a monad. Below we implement co-unit and co-bind with **box** and **let box**.

$$\begin{array}{l}
\epsilon : \square_{\emptyset} \tau \rightarrow \tau \qquad \triangleq \lambda x : \square_{\emptyset} \tau . \mathbf{let box} y = x \mathbf{in} y \\
* : (\square_{\emptyset} \tau_1 \rightarrow \tau_2) \rightarrow \square_{\emptyset} \tau_1 \rightarrow \square_{\emptyset} \tau_2 \qquad \triangleq \lambda f : (\square_{\emptyset} \tau_1 \rightarrow \tau_2) . \lambda x : \square_{\emptyset} \tau_1 . \mathbf{box} (fx)
\end{array}$$

Another significant difference is that variables in the term variable context are annotated with a label¹¹. Examining the static semantics in Figure 15 makes this and some of the other differences clearer.

Firstly, the judgments of $\lambda_{\square\Diamond}$ all have an annotation following the type that is not unlike the effect annotation in λ_{FX} . We write $\Gamma \vdash e : \tau[\ell]$ to mean »term e has type τ and may produce effect ℓ with respect to the context Γ «. As with λ_{\square} , we use an auxiliary typing judgement for expressions. We write $\Gamma \vdash E \div \tau[\ell]$ to mean »expression E has type τ and may product effect ℓ with respect to context Γ «.

Rather than having a specific subsumption rule, subsumption of labels has been pushed into **wft:int**, **wft:var**, **wft:abs**, and **wft:box**. Interestingly, **wft:var** propagates the label on the assumption similar to λ_{SEC} . Another interesting characteristic is that in **wft:abs** functions are required to be pure, in the sense that their argument has no effects and their body must not reveal any effects.

The rule **wft:box** illustrates the role of modal necessity in capturing the effects of an expression. Like **val** in λ_{\square} , **box** suspends the computation of e . As is shown in **wft:letb**, the elimination form for **box** makes sure that the effects e_1 produce are propagated via the label on x in the context of e_2 . However, unlike **let val**, **let box** does not eagerly evaluate the contents of the **box** before substituting. Because this and some of the other operational semantics are non-obvious we present

¹¹Interestingly, Nanevski's account of modal types does not require two term variable context zones.

$$\begin{array}{c}
\frac{}{\mathbf{let\ box\ } x = \mathbf{box\ } e_1 \mathbf{ in\ } e_2 \rightsquigarrow e_2[e_1/x]} \text{ev:letb} \qquad \frac{e \rightsquigarrow e'}{[e] \rightsquigarrow [e']} \text{ev:unit} \\
\frac{E_1 \rightsquigarrow E'_1}{\mathbf{let\ dia\ } x = \mathbf{dia\ } E_1 \mathbf{ in\ } E_2 \rightsquigarrow \mathbf{let\ dia\ } x = \mathbf{dia\ } E'_1 \mathbf{ in\ } E_2} \text{ev:letd1} \\
\frac{}{\mathbf{let\ dia\ } x = \mathbf{dia\ } v \mathbf{ in\ } E \rightsquigarrow E[v/x]} \text{ev:letd2}
\end{array}$$

Figure 16: Selected reductions for λ_{\square} .

a few of the critical reductions in Figure 16. The injection from expressions, **dia**, again acts as a thunk or suspension.

Much like **val** in λ_{\circ} , **dia** captures the effects of the enclosing expression. The unit operator, $[\cdot]$, lifts the effects of the term to be effects of the expression. The monadic bind, **let dia**, threads the effects of the argument through the label on x in the context of E ¹².

2.4.1 Examples

For $\lambda_{\square\Diamond}$ we will actually start with nontermination because it actually proves to be the simplest example.

$$\frac{\Gamma, x : \tau[\ell] \vdash e : \tau[\ell] \quad \text{diverge} \leq \ell}{\Gamma \vdash \mathbf{fix\ } x : \tau . e : \tau[\ell]} \text{wft:fix}$$

Here the typing rule for the fix-point operator looks quite similar to what we used earlier in λ_{FX} . Now consider our stock infinite loop example.

$$\cdot \not\vdash \mathbf{fix\ } x : (\text{int} \rightarrow \text{int})[\text{diverge}] . \lambda y : \text{int}[\emptyset] . (x\ y) : (\text{int} \rightarrow \text{int})[\text{diverge}] \quad (16)$$

The above program is actually ill typed because the use of x in the body of the abstraction will result in the body having a non-empty label. While the overall judgement correctly indicates that the entire expression could diverge, it does not correctly capture that the output of the function is a computation that could also lead to divergence. This can be corrected with an appropriate use of modal necessity.

$$\begin{array}{l}
\cdot \vdash \mathbf{fix\ } x : (\text{int} \rightarrow \square_{\text{diverge}} \text{int})[\text{diverge}] . \\
\lambda y : \text{int}[\emptyset] . \mathbf{box\ } (\mathbf{let\ box\ } z = (x\ y) \mathbf{ in\ } z) : (\text{int} \rightarrow \square_{\text{diverge}} \text{int})[\text{diverge}]
\end{array} \quad (17)$$

It is interesting to note that this is almost the exact reverse of Program Fragment 15, where we first must bind the recursive application and then wrap the result back up in a unit. Of course, we could

¹²I had to massage Nanevski's rules for expression some in order for my reference cell examples to work. Does this indicate a mismatch between using labels versus his names?

reverse the order of operations and the behavior and type would remain unchanged. Changing the order would not be nearly as trivial in λ_{\circ} ; this is the first hint that necessity provides us some additional freedoms over the lax modality.

Exceptions in $\lambda_{\square\Diamond}$ are slightly more interesting because they are an effect that can be handled.

$$\frac{\text{exn}(i) \leq \ell}{\Gamma \vdash \mathbf{throw\ exn}(i) : \tau[\ell]} \text{wft:throw} \quad \frac{\Gamma \vdash e_1 : \tau[\ell_1] \quad \Gamma \vdash e_2 : \tau[\ell_2] \quad \ell_2 \leq \ell_1 \bullet \text{exn}(i)}{\Gamma \vdash \mathbf{catch\ exn}(i)\ \mathbf{with}\ e_1\ \mathbf{in}\ e_2 : \tau[\ell_1]} \text{wft:catch}$$

Again, the typing rules for **throw** and **catch** are remarkably similar to those in λ_{FX} . However, unlike λ_{FX} functions must remain pure and therefore if we write a function that may throw an exception we need to use modal necessity to properly package the effects. For example, this program

$$\text{halve} \triangleq \cdot \vdash \lambda x : \text{int} . \mathbf{box\ (if\ } x = 0 \mathbf{\ then\ (throw\ exn(divbyzero))\ : int} \rightarrow \square_{\text{exn}(divbyzero)} \text{int}[\emptyset] \mathbf{else\ (x/2)} \tag{18}$$

will encapsulate the results in a **box**. To use this the result of this function we would need to use **let box**

$$\cdot \vdash \mathbf{let\ box\ } x = \text{halve } 42 \mathbf{\ in\ (x + x) : int}[\text{exn}(divbyzero)] \tag{19}$$

However, this will actually result in the conditional being evaluated twice because of the the call-by-name nature of **let box**. But this provides another indication of the flexibility necessity buys us: we are not restricted to strict single-threading and sequentialization of computations. Note that the effect has been propagated as expected. If we wanted to mask the effect we can use **catch** to eliminate it

$$\cdot \vdash \mathbf{let\ box\ } x = \text{halve } 42 \mathbf{\ in\ (catch\ exn}(divbyzero) \mathbf{\ with\ } 0 \mathbf{\ in\ (x + x)) : int}[\emptyset] \tag{20}$$

Naïvely we might continue as we have and give reference cells the following static semantics

$$\frac{\Gamma \vdash e : \tau[\ell]}{\Gamma \vdash \mathbf{ref}^l e : \mathbf{ref}^l \tau[\ell \bullet \text{new}(l)]} \text{wft:ref} \quad \frac{\Gamma \vdash e : \mathbf{ref}^l \tau[\ell]}{\Gamma \vdash !e : \tau[\ell \bullet \text{read}(l)]} \text{wft:deref}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{ref}^l \tau[\ell_1] \quad \Gamma \vdash e_2 : \tau[\ell_2]}{\Gamma \vdash e_1 := e_2 : \mathbf{ref}^l \tau[\ell_1 \bullet \ell_2 \bullet \text{write}(l)]} \text{wft:assn}$$

Except that we immediately run into some unexpected behavior

$$\cdot \vdash \lambda x : \text{int}[\emptyset] . \mathbf{let\ box\ } y = (\mathbf{ref}^l x) \mathbf{\ in\ box\ } \langle y, y \rangle : \text{int} \rightarrow \square_{\text{new}(l)} (\mathbf{ref}^l \text{int} \times \mathbf{ref}^l \text{int})[\emptyset] \tag{21}$$

We might have thought that evaluating this function would return a pair of pointers to the same reference cell, but due to the call-by-name evaluation of **let box**, we actually allocate two difference cells with the same contents. Or worse consider

$$\cdot \vdash \lambda x : \text{int}[\emptyset] . \lambda y : \text{int}[\emptyset] . \mathbf{let\ box\ } z = (\mathbf{ref}^l x) \mathbf{\ in\ let\ box\ } w = (z := y) \mathbf{\ in\ (box\ } \langle z, w \rangle) : \text{int} \rightarrow \square_{\text{new}(l)} (\mathbf{ref}^l \text{int} \times \mathbf{ref}^l \text{int})[\emptyset] \tag{22}$$

Here we would expect to get back two identical reference cells, containing y , but we'll get two distinct reference cells containing x and y respectively. The problem here is that the order in which memory operations occur is important!

One solution would be to simply rewrite these functions so that everything happens in the correct order. However, instead of leaving the programmer to enforce sequentiality, we can have the type system guarantee that the program is threaded correctly using modal possibility.

$$\frac{\Gamma \vdash e : \tau[\ell]}{\Gamma \vdash \mathbf{ref}^l e \div \mathbf{ref}^l \tau[\ell \bullet \mathbf{new}(\ell)]} \text{wfe:ref} \qquad \frac{\Gamma \vdash e : \mathbf{ref}^l \tau[\ell]}{\Gamma \vdash !e \div \tau[\ell \bullet \mathbf{read}(\ell)]} \text{wfe:deref}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{ref}^l \tau[\ell] \quad \Gamma \vdash e_2 : \tau[\ell]}{\Gamma \vdash e_1 := e_2 \div \mathbf{ref}^l \tau[\ell \bullet \mathbf{write}(\ell)]} \text{wfe:assn}$$

These rules are reminiscent of those that we used for λ_{\circ} . We can then repair our above example by using possibility instead of necessity

$$\cdot \vdash \lambda x : \text{int}[\emptyset] . \mathbf{dia} (\mathbf{let} \mathbf{dia} \ y = \mathbf{dia} (\mathbf{ref}^l \ x) \ \mathbf{in} \ [\langle y, y \rangle]) : \text{int} \rightarrow \diamond_{\mathbf{new}(\ell)} (\mathbf{ref}^l \ \text{int} \times \mathbf{ref}^l \ \text{int})[\emptyset] \quad (23)$$

2.5 The linear λ -calculus

Linear logic was originally developed out of research into domain theory and coherence semantics [19]. However, linear logic has since been recognized as a natural language for communicating statements about state and resources. It was only natural that computer scientists would consider the computational interpretation of linear logic to model stateful computation and resources in programming.

At its core, the linear λ -calculus is the same as the simply-typed λ -calculus except that the, often implicit and unstated, structural rules of logic that allowing weakening and contraction of assumptions in contexts have been forbidden.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma, x : \tau_2 \vdash e : \tau_1} \text{weakening} \qquad \frac{\Gamma, x : \tau_1, y : \tau_1 \vdash e : \tau_2}{\Gamma, x : \tau_1 \vdash e : \tau_2} \text{contraction}$$

These two rules are sensible when hypotheses are considered »truths«. For contraction, if you have proofs x and y that τ_1 is true, then anywhere you used the proof y you could have interchangeably used x ¹³. Similarly for weakening, if we could prove τ_1 with the assumptions in Γ then adding another truth τ_2 shouldn't affect the proof.

However, when we think of hypothesis as »resources« the rules become suspect. Contraction, for example, can be interpreted as allowing programs to implicitly copy or alias resources. Weakening states that a program can implicitly choose to destroy a resource. These two structural rules make it difficult to precisely characterize a program's treatment of resources. Therefore, by removing them from our type system, programs are forced to be significantly more explicit about their use of resources to remain well-typed.

Unfortunately, as a programming language the linear λ -calculus proves to be very restrictive: it is only capable of expressing those functions that are computable in linear time¹⁴. To remedy

¹³However, this seems to require a commitment to proof irrelevance.

¹⁴It would be interesting to investigate whether they are known to correspond exactly to the same complexity class as deterministic finite automata. Martin Hofmann's work might say something about this [22].

Types	$\tau ::= \top \mid \text{int} \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid \tau_1 \oplus \tau_2 \mid !\tau \mid \dots$
Terms	$e ::= * \mid \text{let } * = e_1 \text{ in } e_2 \mid i \mid x \mid \lambda x : \tau . e$ $\mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \text{let } \langle x, y \rangle = e_1 \text{ in } e_2$ $\mid \text{inl } e \mid \text{inr } e \mid \text{case } e_1 \text{ of inl } x \Rightarrow e_2, \text{inr } y \Rightarrow e_3$ $\mid \text{derelict } e \mid \text{discard } e_1 \text{ in } e_2 \mid \text{copy } e_1 \text{ as } x, y \text{ in } e_2$ $\mid \text{promote } e_1, \dots, e_n \text{ for } x_1, \dots, x_n \text{ in } e \mid \dots$
Term variable context	$\Gamma ::= \cdot \mid \Gamma, x : \tau$

Figure 17: The grammar for the linear λ -calculus.

this, it is common to add back some of the power provided by weakening and contraction in a controlled fashion. This is done by introducing the idea of an *unrestricted* resource. A specific integer resource would be given type `int`; an integer that may be used any number of times will be given type `!int` where `!` is a modal operator pronounced »of course« or »bang«. It is important to note that like $\square, !$ forms a comonad; this will come back later.

Operationally, we may think of unrestricted resources as resources allocated on the heap that must be managed by a garbage collector (because we cannot statically decide whether they are aliased). Restricted resources behave more a little more like memory allocated in a language like `C`, where we must specifically manage their allocation and cleanup. However, unlike memory allocated in `C`, the typing rules of the linear λ -calculus prevent the aliasing of restricted resources.

The grammar for the linear λ -calculus extended with the `!` modality, multiplicative units and products, and additive sums is given in Figure 17. The different term formulations of the linear λ -calculus are legion [27, 2, 43, 37, 10] (to cite just a few). The one we have chosen is due to Benton et al. [8]. This particular formulation seemed ideal because Benton and Wadler have shown how to relate it to λ_{O} via adjoint models [9]. We will explore this relationship in more detail in Section 2.6.

As is traditional, functions types in the linear λ -calculus are written $\tau_1 \multimap \tau_2$ using the »lolly« or multimap operator. \top is the type of the »multiplicative« unit¹⁵ and we use the tensor product to indicate $\tau_1 \otimes \tau_2$ multiplicative product types. The additive sum type \oplus behaves essentially the same as the traditional disjoint sum type. Informally, when constructing additive connectives we are allowed share resources among their premises, whereas multiplicative connectives must split their resources. The multiplicatives are sometimes said to be »conservative« over their contexts [19].

The static semantics of our version of the linear λ -calculus is sketched in Figure 18. The fact that `wft:unit` requires that there be no hypotheses has to do with the intuition that any resources flowing into a term through the hypotheses must be consumed. This is something like a conservation law in physics – all resources that go in, must come out¹⁶. Because no resources are necessary to construct `*`, there must be no hypotheses. Otherwise it would be possible to use the creation of `*`

¹⁵It is useful to note that there is some inconsistency in the literature with regards to the notation for units. For example, Pfenning writes `1` for the multiplicative unit and \top for the additive unit [37]. Others, such as Barr and Wells write \top for the multiplicative unit and `1` for the additive unit [6]. This can be attributed to the fact that they would like to associate the additive with terminal objects in a category. Because we make similar categorical connections, we follow the latter choice of notation.

¹⁶In the presence of additive units this analogy does not seem as apt, but I still think it provides a useful intuition

$$\begin{array}{c}
\frac{}{\cdot \vdash * : \top} \text{wft:unit} \qquad \frac{\Gamma_1 \vdash e_1 : \top \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathbf{let} * = e_1 \mathbf{in} e_2 : \tau} \text{wft:unite} \qquad \frac{}{\cdot \vdash i : \text{int}} \text{wft:int} \\
\\
\frac{}{\cdot, x : \tau \vdash x : \tau} \text{wft:var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \multimap \tau_2} \text{wft:abs} \\
\\
\frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \quad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : \tau_2} \text{wft:app} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2} \text{wft:pair} \\
\\
\frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \quad \Gamma_2, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathbf{let} \langle x, y \rangle = e_1 \mathbf{in} e_2 : \tau} \text{wft:proj} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{inl} e : \tau_1 \oplus \tau_2} \text{wft:inl} \\
\\
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{inr} e : \tau_1 \oplus \tau_2} \text{wft:inr} \qquad \frac{\Gamma_1 \vdash e : \tau_1 \oplus \tau_2 \quad \Gamma_2, x : \tau_1 \vdash e_2 : \tau \quad \Gamma_2, y : \tau_2 \vdash e_3 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathbf{case} e_1 \mathbf{of} \mathbf{inl} x \Rightarrow e_2, \mathbf{inr} y \Rightarrow e_3 : \tau} \text{wft:case} \\
\\
\frac{\Gamma \vdash e : !\tau}{\Gamma \vdash \mathbf{derelict} e : \tau} \text{wft:der} \qquad \frac{\Gamma_1 \vdash e_1 : !\tau_1 \quad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash \mathbf{discard} e_1 \mathbf{in} e_2 : \tau_2} \text{wft:dis} \\
\\
\frac{\Gamma_1 \vdash e_1 : !\tau_1 \quad \Gamma_2, x : !\tau_1, y : !\tau_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash \mathbf{copy} e_1 \mathbf{as} x, y \mathbf{in} e_2 : \tau_2} \text{wft:copy} \\
\\
\frac{\Gamma_i \vdash e_i : !\tau_i \quad x_1 : !\tau_1, \dots, x_n : !\tau_n \vdash e : \tau}{\Gamma_1, \dots, \Gamma_n \vdash \mathbf{promote} e_1, \dots, e_n \mathbf{for} x_1, \dots, x_n \mathbf{in} e : !\tau} \text{wft:prom}
\end{array}$$

Figure 18: The static semantics of the linear λ -calculus

to » destroy « resources. Interestingly, unlike standard Intuitionistic unit, multiplicative units have an elimination form as specified in `wft:unite`.

Much like `wft:unit`, the typing rule for variables requires that the variable be the sole assumption. The typing rule for abstraction remains unchanged. Application is essentially unchanged as well, except that we must explicitly divide the contents of the context between the function and the argument. Again for the multiplicative product we must explicitly divide the assumptions between the components. Unlike standard products, multiplicative products must have both components projected simultaneously to ensure the conservation of resources. Injection and case dispatch for additive sums are essentially identical to that for disjoint sums in standard presentations of typed λ -calculus.

There are four different terms for explicitly manipulating unrestricted resources. The dereliction term, **derelict** e , allows for converting an unrestricted resource into a single use, restricted resource. The discard term, **discard** e_1 **in** e_2 , allows the current context to forget about the existence of an unrestricted resource – much like weakening, but made explicit. To understand why this is sound, it is useful to think about the operational intuition where unrestricted resources are managed by a garbage collector rather than the program itself. Since we are not managing the resource ourselves, it is safe for us to forget about its existence. The copy term, **copy** e_1 **as** x, y **in** e_2 , provides a way to duplicate unrestricted resources – much like contraction. While called copy, for our purposes the intuition should be that it really just creates aliases to memory locations. Finally, promotion, **promote** e_1, \dots, e_n **for** x_1, \dots, x_n **in** e , says that if we can construct a term, e , solely from unrestricted resources, e_1, \dots, e_n , then it too can be treated as unrestricted.

2.5.1 Examples

The linear λ -calculus has quite a different flavor from all the other systems we have examined so far. Consider a simple implementation of reference cells

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}^l e : \mathbf{ref}^l \tau} \text{ wft:ref} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{ref}^l \tau \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash e_1 :=: e_2 : \mathbf{ref}^l \tau \otimes \tau} \text{ wft:swap}$$

The rule for allocation looks normal, but instead of having operations for reading and writing reference cells, we have a single operation, `:=:`, called » swap «. This is because allowing arbitrary reads to a reference cell would result in unexpected duplication of resources, and similarly assignment would allow for the destruction of resources. Therefore, `swap` allows the contents of a memory cell to be read and written while conserving the overall resources in the system. However, for reference cells to be particularly interesting in the linear λ -calculus, we generally need to make them unrestricted; aliasing is what makes references interesting!

While seemingly more restrictive, this implementation of reference cells, along with the structural rules for manipulating unrestricted resources, can be used to implement all the functionality found in our earlier presentations of reference cells. Our implementation uses the type $!\mathbf{ref}^l (!\tau \oplus \top)$ for one of these derived reference cells. The reason we need the disjoint sum, is that in order to do a read, we will need to swap out the contents of the memory cell. However, we may not have anything of the appropriate type to swap into the cell handy. So instead we allow for cells to be optionally empty by swapping in unit. We can then implement allocation, reading, and writing as

the following

$$\begin{aligned}
\text{ref} & : !\tau \multimap !\text{ref}^l (!\tau \oplus \top) & = \lambda x : !\tau . \mathbf{promote} \ x \ \mathbf{for} \ y \ \mathbf{in} \ \text{ref}^l (\text{inl } y) \\
\text{deref} & : !\text{ref}^l (!\tau \oplus \top) \multimap !\tau & = \lambda x : !\text{ref}^l (!\tau \oplus \top) . \\
& & \quad \mathbf{let} \ \langle y, z \rangle = ((\mathbf{derelict} \ x) :=: \mathbf{inr} \ *) \ \mathbf{in} \\
& & \quad (\mathbf{case} \ z \ \mathbf{of} \ \mathbf{inl} \ u \Rightarrow \\
& & \quad \quad (\mathbf{copy} \ u \ \mathbf{as} \ l, m \ \mathbf{in} \\
& & \quad \quad \quad \mathbf{let} \ \langle s, t \rangle = ((\mathbf{derelict} \ x) :=: \mathbf{inl} \ l) \ \mathbf{in} \\
& & \quad \quad \quad \mathbf{case} \ t \ \mathbf{of} \ \mathbf{inl} \ p \Rightarrow \mathbf{abort}, \\
& & \quad \quad \quad \mathbf{inr} \ q \Rightarrow (\mathbf{let} \ * = q \ \mathbf{in} \\
& & \quad \quad \quad \quad \mathbf{discard} \ (\mathbf{promote} \ \cdot \ \mathbf{for} \ \cdot \ \mathbf{in} \ s) \ \mathbf{in} \ m)), \\
& & \quad \quad \mathbf{inr} \ v \Rightarrow \mathbf{abort}) \\
\text{assn} & : !\text{ref}^l (!\tau \oplus \top) \otimes !\tau \multimap !\text{ref}^l (!\tau \oplus \top) & = \lambda x : !\text{ref}^l (!\tau \oplus \top) \otimes !\tau . \\
& & \quad \mathbf{let} \ \langle y, z \rangle = x \ \mathbf{in} \\
& & \quad \mathbf{promote} \ y, z \ \mathbf{for} \ p, q \ \mathbf{in} \\
& & \quad \mathbf{let} \ \langle u, v \rangle = ((\mathbf{derelict} \ p) :=: \mathbf{inl} \ q) \ \mathbf{in} \\
& & \quad (\mathbf{case} \ v \ \mathbf{of} \ \mathbf{inl} \ l \Rightarrow (\mathbf{discard} \ l \ \mathbf{in} \ u), \\
& & \quad \quad \mathbf{inr} \ m \Rightarrow \mathbf{abort})
\end{aligned}$$

The complexity of these functions illustrates just how much memory management is hidden in a naïve implementation of reference cells^{17 18}. Notice the use of dereliction, discarding, and promotion required to access and create unrestricted resources as well as free up unused aliases.

The implementation of allocation is fairly straightforward, because we are constructing a reference cell solely from the unrestricted resource we provided, we are allowed to promote the entire reference cell to be unrestricted. Dereference is quite complicated, firstly, we must swap in a multiplicative unit to read out the contents of the cell. Then we must copy the contents, swap one of the copies back into the cell, promote the lingering reference to the cell so that we can then discard it before returning the actual contents. Implementing assignment is actually easier than dereference because only one swap is necessary.

We cheated slightly by using a construct **abort** in the »impossible« cases.

$$\frac{}{\Gamma \vdash \mathbf{abort} : \tau} \text{wft:abort}$$

Operationally, **abort** simply stops execution of the program, so it is acceptable for it to consume all resources and appear to be of any type¹⁹. **abort** would be unnecessary if we were to enrich the type system in some fashion to capture the invariant that our derived reference cells are never empty outside of the deref function.

We can also look at how nontermination is expressed in the linear λ -calculus. There are actually two variations on the fix-point operator we can consider.

$$\frac{!\Gamma, x : !\tau \vdash e : !\tau}{!\Gamma \vdash \mathbf{fix} \ x : !\tau . e : !\tau} \text{wft:fix} \qquad \frac{!\Gamma, x : \tau \vdash e : \tau}{!\Gamma \vdash \mathbf{fix} \ x : \tau . e : \tau} \text{wft:fix-lin}$$

¹⁷It took several tries to get correct, though this is just a conjecture because I do not have a typechecker.

¹⁸Actually I'm still not sure deref is correct. Check this.

¹⁹Perhaps it might be useful to think of an additive unit as a lazy abort – it does not immediately halt execution, but we cannot eliminate it.

We write $!\Gamma$ to mean that $\forall x : \tau \in \Gamma, \tau = !\tau'$ for some τ' . The **wft:fix** rule requires that all resources available inside the fix-point be unrestricted. This requirement is necessary because in the presence of general recursion it is impossible to decide statically whether resources will be used linearly or not. So instead we are conservative and require all resources to be unrestricted. The second rule, **wft:fix-lin**, allows the fix-point itself to be used linearly. However, a non-obvious consequence of this is that any use of **wft:fix-lin** will always diverge, because we must always recurse in order to consume the resource bound to x .

To extend the linear λ -calculus with exceptions, we must make similar restrictions.

$$\frac{}{!\Gamma \vdash \mathbf{throw} \text{exn}(i) : \tau} \text{wft:throw} \qquad \frac{!\Gamma \vdash e_1 : \tau \quad !\Gamma \vdash e_2 : \tau}{!\Gamma \vdash \mathbf{catch} \text{exn}(i) \mathbf{with} e_1 \mathbf{in} e_2 : \tau} \text{wft:catch}$$

Because **throw** can abandon the current control-flow context, we must require that all resources are unrestricted, because there will not be an opportunity to »clean-up« restricted resources before jumping to the exception handler²⁰. The **catch** term must also require that all resources be unrestricted, this is because there is no guarantee the exception handler e_1 will run and consume any of its resources, and we cannot be sure that e_2 will finish executing and consume all of its resources either.

Aside from providing a precise account of resource management in programs, linearity can also be used to track dependencies. Imagine extending our language with constants that act as term representations of the labels we used in previous systems. We could then make the typing rule for our running password example:

$$\frac{}{\cdot \vdash \text{password} : \text{int} \otimes \mathbf{secret}} \text{wft:pw}$$

Here the password becomes a multiplicative product of the password's value plus a token for its security level. In order for any piece of code to get at the password's value it must first destruct the product:

$$\cdot \vdash \mathbf{let} \langle x, y \rangle = \text{password} \mathbf{in} \dots : \tau \tag{24}$$

However, the strict resource discipline we impose provides no way for the code to destroy the token for **secret**, and therefore is forced to carry it along through all the remaining computations. Consequently, **secret** will be forced to appear somewhere in the type of τ , flagging it as depending upon high security data. For example,

$$\cdot \vdash \mathbf{let} \langle x, y \rangle = \text{password} \mathbf{in} \langle x + 1, y \rangle : \text{int} \otimes \mathbf{secret} \tag{25}$$

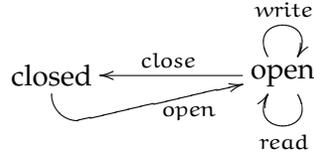
$$\cdot \vdash \mathbf{let} \langle x, y \rangle = \text{password} \mathbf{in} \lambda z : \text{int} . \langle z + x, y \rangle : \text{int} \multimap (\text{int} \otimes \mathbf{secret}) \tag{26}$$

While interesting from a comparison standpoint, we do not think that actually implementing information-flow tracking in an source language via linearity is likely to be practical. It might make sense in an intermediate language, however. Particularly if information-flow policies were being implemented and enforced cryptographically.

Finally, another interesting application of linearity is to encode finite state protocols in the type system [14, 28]. A canonical example is a file-handle. Ideally programs would adhere to a protocol

²⁰It might be interesting to formalize such a thing, however; it would be something like finalizers.

where they must first open a file-handle before they can read and write to it, and then when they are finished they must close the file-handle to make sure the resource is returned to the system. Additionally, we would like to prevent a file-handle from being opened twice consecutively without an intervening close. We can represent this through the following automata diagram:



The state transitions of this automaton can be cleanly encoded using the type system of the linear λ -calculus:

```

new   :  $\top \multimap \text{fhandle}(\text{closed})$ 
open  :  $\text{string} \multimap \text{fhandle}(\text{open})$ 
read  :  $(\text{fhandle}(\text{open}) \otimes \text{int}) \multimap (\text{fhandle}(\text{open}) \otimes \text{string})$ 
write :  $(\text{fhandle}(\text{open}) \otimes \text{string}) \multimap \text{fhandle}(\text{open})$ 
close :  $\text{fhandle}(\text{open}) \multimap \text{fhandle}(\text{closed})$ 
free  :  $\text{fhandle}(\text{closed}) \multimap \top$ 
  
```

While some bits of this protocol can be encoded in a typical type system, because of aliasing it is impossible to guarantee that there will be no attempt to read a closed file-handle, and that all file-handles will be freed eventually.

2.6 LNL and adjoint models

So far we have explored a number of different ways of tracking effects and dependencies in a static type system. Most of them had a very similar flavor: λ_{FX} , λ_{SEC} , λ_{\circ} , $\lambda_{\square\lozenge}$ all used some kind of labeling system. Both λ_{\circ} and $\lambda_{\square\lozenge}$ demarcated and threaded effects and dependencies using modalities. The linear λ -calculus seemed to be the only radically different system we examined. Still given that all these type systems seem capable of expressing and enforcing similar kinds of properties about programs, it is natural to ask whether there is underlying relationship between all of them.

As was evident from some of our examples of programming with the linear λ -calculus, it can require quite a bit of tedious explicit management of resources. It might be more practical if there was a way to write programs as normal, but be able to avail oneself of linearity when useful. Research by Benton into developing a more practical language for balancing the benefits and disadvantages of linearity led to the discovery of a model for the linear λ -calculus that can also act as a model for λ_{\circ} [15]. Benton called the logic underlying this model Linear/Non-linear Logic (LNL) because there is a separation between the linear and traditional Intuitionistic computational paradigms mediated by two special operators [7]. Benton and Wadler later called the term language for this logic the » adjoint calculus « [9].

2.6.1 Further categorical foundations

To explain the adjoint model Benton developed, we must first make another digression into category theory.

Definition 2.5 An adjoint model is composed of:

- A Cartesian closed category $\langle \mathcal{C}, 1, \times, \rightarrow \rangle$,
- A symmetric monoidal closed category $\langle \mathcal{L}, \top, \otimes, -\circ \rangle$,
- A symmetric monoidal adjunction $\langle F, G, \eta, \epsilon, m, n \rangle$ from \mathcal{C} to \mathcal{L} .

A Cartesian closed category is a category with all finite limits and exponentials²¹, and are generally well known because they are natural models of equational theories on the simply typed λ -calculus. The category of sets and functions between them is the classic example of a Cartesian closed category.

Symmetric monoidal categories A symmetric monoidal closed category is a bit more exotic structure, so we will explain in it more detail here. It is useful to begin by describing a monoidal category.

Definition 2.6 Monoidal categories are categories \mathcal{C} equipped with:

- A distinguished object \top .
- A bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$.
- A natural isomorphism $\alpha : X \otimes (Y \otimes Z) \xrightarrow{\sim} (X \otimes Y) \otimes Z$ for all objects $X, Y, Z \in \mathcal{C}$.
- A natural isomorphism $\lambda : \top \otimes X \xrightarrow{\sim} X$ for all objects $X \in \mathcal{C}$.
- A natural isomorphism $\rho : X \otimes \top \xrightarrow{\sim} X$ for all objects $X \in \mathcal{C}$.

such that the following diagrams commute

$$\begin{array}{ccc}
 & X \otimes (Y \otimes (Z \otimes W)) & \\
 \swarrow \scriptstyle X \otimes \alpha(Y, Z, W) & & \searrow \scriptstyle \alpha(X, Y, Z \otimes W) \\
 X \otimes ((Y \otimes Z) \otimes W) & & (X \otimes Y) \otimes (Z \otimes W) \\
 \downarrow \scriptstyle \alpha(X, Y \otimes Z, W) & & \downarrow \scriptstyle \alpha(X \otimes Y, Z, W) \\
 (X \otimes (Y \otimes Z)) \otimes W & \xrightarrow{\scriptstyle \alpha(X, Y, Z) \otimes W} & ((X \otimes Y) \otimes Z) \otimes W
 \end{array}$$

$$\begin{array}{ccc}
 X \otimes (\top \otimes Y) & \xrightarrow{\scriptstyle \alpha(X, \top, Y)} & (X \otimes \top) \otimes Y \\
 \searrow \scriptstyle X \otimes \lambda(Y) & & \swarrow \scriptstyle \rho(X) \otimes Y \\
 & X \otimes Y &
 \end{array}$$

These diagrams are mostly just an elaborate way of saying that \otimes is associative, and that \top behaves as a unit should.

²¹One of many other alternative definitions is a category with terminal objects, Cartesian products, and exponentials.

Definition 2.7 A symmetric monoidal category builds on a monoidal category by requiring a natural isomorphism $\sigma : X \otimes Y \xrightarrow{\sim} Y \otimes X$ for all objects $X, Y \in \mathcal{C}$, such that the following diagrams commute

$$\begin{array}{ccc}
 X \otimes Y & \xrightarrow{\sigma(X,Y)} & Y \otimes X \\
 \searrow \text{id}_{X \otimes Y} & & \swarrow \sigma(Y,X) \\
 & X \otimes Y &
 \end{array}
 \quad
 \begin{array}{ccc}
 X \otimes T & \xrightarrow{\sigma(X,T)} & T \otimes X \\
 \searrow \lambda(X) & & \swarrow \rho(X) \\
 & X &
 \end{array}$$

$$\begin{array}{ccc}
 X \otimes (Y \otimes Z) & \xrightarrow{\alpha(X,Y,Z)} & (X \otimes Y) \otimes Z \\
 \downarrow X \otimes \sigma(Y,Z) & & \downarrow \sigma(X \otimes Y, Z) \\
 X \otimes (Z \otimes Y) & & Z \otimes (X \otimes Y) \\
 \downarrow \alpha(X,Z,Y) & & \downarrow \alpha(Z,X,Y) \\
 (X \otimes Z) \otimes Y & \xrightarrow{\sigma(X,Z) \otimes Y} & (Z \otimes X) \otimes Y
 \end{array}$$

These diagrams codify what it means for \otimes to be a commutative operation.

Adjunctions and symmetric monoidal closed categories Finally, to explain what makes a symmetric monoidal category »closed« we feel we should briefly review the notion of adjunctions. Adjunctions are common in category theory, in fact they are considered to be of fundamental importance, but they still tend to be beyond the ken of a novice.

Definition 2.8 An adjunction is a structure over two categories \mathcal{C} and \mathcal{D} consisting of

- A pair of functors $F : \mathcal{C} \rightarrow \mathcal{D}$ and $G : \mathcal{D} \rightarrow \mathcal{C}$.
- A natural transformation $\eta : \text{Id}_{\mathcal{C}} \xrightarrow{\sim} (G \circ F)$.

such that for each object X and arrow $f : X \rightarrow G(Y)$ in \mathcal{C} there is a unique arrow $f^* : F(X) \rightarrow Y$ such that this diagram commutes

$$\begin{array}{ccc}
 X & \xrightarrow{\eta_X} & G(F(X)) \\
 \searrow f & & \downarrow G(f^*) \\
 & & G(Y)
 \end{array}$$

The natural transformation η is called the unit of the adjunction. All adjunctions are also equipped with a co-unit, $\epsilon : (F \circ G) \xrightarrow{\sim} \text{Id}_{\mathcal{D}}$, that is inter-definable with the unit. F is called the *left-adjoint* of G , and G is called the *right-adjoint* of F . It is not uncommon for a novice to have no intuition whatsoever as to what this definition means, so we will hopefully clarify with some examples.

One of the simplest examples we can consider is a when $F : \mathcal{C} \rightarrow \mathbf{1}$ where $\mathbf{1}$ is the terminal category with a single object $*$ and only the identity morphism. The right adjoint of F is then a

functor $G : \mathbf{1} \rightarrow \mathcal{C}$.

$$\begin{array}{ccc}
 X & \xrightarrow{\eta_X} & G(F(X)) \\
 & \searrow f & \downarrow G(f^*) \\
 & & G(Y)
 \end{array}
 \Rightarrow
 \begin{array}{ccc}
 X & \xrightarrow{!_X} & X' \\
 & \searrow f & \downarrow \text{id}_{X'} \\
 & & X'
 \end{array}$$

The category $\mathbf{1}$ has a only single object, $*$, therefore $Y = F(X) = *$. Consequently, $G(Y) = G(F(X)) = X'$ for some object X' in \mathcal{C} . There is only a single morphism in $\mathbf{1}$, so $f^* = \text{id}_*$. Functors preserve identities, so $G(\text{id}_*) = \text{id}_{X'}$. Furthermore, $\text{id}_{X'} \circ \text{id}_{X'} = \text{id}_{X'}$. Therefore, if the above diagram commutes, then any morphism $f : X \rightarrow X'$ in \mathcal{C} must be identical to the unit $!_X$, which means that for a given object X there is a unique morphism to X' . Furthermore, regardless of which X we pick, G must always map to the same X' ²². Consequently, X' has all the properties of a terminal object. Therefore, if there exists a right-adjoint G for the functor $F : \mathcal{C} \rightarrow \mathbf{1}$ then the category \mathcal{C} has a terminal object.

Another important and relevant example of an adjunction are exponentials in Cartesian closed categories.

$$\begin{array}{ccc}
 F(G(Y)) & \xrightarrow{\epsilon_Y} & Y \\
 \uparrow F(g^*) & \nearrow g & \\
 F(X) & &
 \end{array}
 \Rightarrow
 \begin{array}{ccc}
 Y^Z \times Z & \xrightarrow{\text{eval}_{ZY}} & Y \\
 \uparrow \text{curry} \times \text{id}_Z & \nearrow g & \\
 X \times Z & &
 \end{array}$$

Here we have taken the $F : \mathcal{C} \rightarrow \mathcal{C}$ to be the functor $- \times Z$, making its right-adjoint $G : \mathcal{C} \rightarrow \mathcal{C}$ the exponential functor $-^Z$. The co-unit is evaluation with respect to Z and the unique arrow guaranteed by the adjunction corresponds to currying g . Therefore, we can approximately understand this right-adjoint as internalizing the morphisms of the category.

In general, it is helpful think of an adjunction as forming a sort of bijection between structures, thought not necessarily an equivalence. Another intuition is that adjunctions are useful in showing how to construct objects with a universal property (such as the terminal object or exponentials) by relating a category to another category that possesses the structure of interest.

Finally, it is important to note that adjunctions naturally form monadic and comonadic structures; the use of the same notation and naming for units and co-units is not merely a coincidence.

Returning to where we started, we were seeking an understanding of symmetric monoidal closed categories. A symmetric monoidal category is *closed* if every functor $- \otimes X$ has a right-adjoint. We will generally write this adjoint as $X \multimap Y$.

Given how closely the the definitions of Cartesian closed categories and symmetric monoidal closed categories coincide, it is natural to wonder just how they differ. The key difference is that \top and \otimes are not universal constructions in the same sense as the terminal object, 1 , and the Cartesian product, \times [42]. For example, there are not necessarily any projections from \otimes , nor is there guaranteed to be a unique arrow from all objects to \top . However, if for a given symmetric monoidal closed category, \mathcal{C} , we take \top and \otimes to be 1 and \times , respectively, then \mathcal{C} is a Cartesian closed category.

²²This is not clear enough on how it is guaranteed that every X has a morphism to X' .

The canonical example of a symmetric monoidal closed category is the category of coherence domains with linear maps as morphisms. Another less esoteric example is the category of Abelian groups and group homomorphisms. Finally, symmetric monoidal closed categories are used as semantic models of the linear λ -calculus.

Symmetric monoidal functors The last piece of Definition 2.5 requires that there be a symmetric monoidal adjunction $\langle F, G, \eta, \epsilon, \mu, \nu \rangle$ from \mathcal{C} to \mathcal{L} . That is F is a symmetric monoidal functor from \mathcal{C} to \mathcal{L} with the unit μ_1 and the natural transformation $\mu_{X,Y}$ and G is a symmetric monoidal functor from \mathcal{L} to \mathcal{C} with the unit ν_1 and the natural transformation $\nu_{X,Y}$.

Definition 2.9 A functor F between two symmetric monoidal categories $\langle \mathcal{M}, \otimes, \top, \alpha, \lambda, \rho \rangle$ and $\langle \mathcal{M}', \otimes', \top', \alpha', \lambda', \rho' \rangle$ is symmetric monoidal if it comes equipped with

- A map $\mu_\top : \top' \rightarrow F(\top)$ in \mathcal{M}'
- A natural transformation $\mu_{X,Y} : F(X) \otimes' F(Y) \rightarrow F(X \otimes Y)$ for $X, Y \in \mathcal{M}$.

Furthermore, the following diagrams are required to commute

$$\begin{array}{ccc}
 (F(X) \otimes' F(Y)) \otimes' F(Z) & \xrightarrow{\alpha'(X,Y,Z)} & F(X) \otimes' (F(Y) \otimes' F(Z)) \\
 \downarrow \mu_{X,Y} \otimes' \text{Id} & & \downarrow \text{Id} \otimes' \mu_{Y,Z} \\
 F(X \otimes Y) \otimes' F(Z) & & F(X) \otimes' F(Y \otimes Z) \\
 \downarrow \mu_{X \otimes Y, Z} & & \downarrow \mu_{X, Y \otimes Z} \\
 F((X \otimes Y) \otimes Z) & \xrightarrow{F(\alpha(X,Y,Z))} & F(X \otimes (Y \otimes Z))
 \end{array}$$

$$\begin{array}{ccccc}
 \top' \otimes' F(X) & \xrightarrow{\lambda'} & F(X) & & F(X) \otimes' \top' & \xrightarrow{\rho'} & F(X) & & F(X) \otimes' F(Y) & \xrightarrow{\sigma'} & F(Y) \otimes' F(X) \\
 \mu_\top \otimes' \text{Id} \downarrow & & \uparrow F(\lambda) & & \text{Id} \otimes' \mu_\top \downarrow & & \uparrow F(\rho) & & \mu_{X,Y} \downarrow & & \mu_{Y,X} \downarrow \\
 F(\top) \otimes' F(X) & \xrightarrow{\mu_{\top,X}} & F(\top \otimes X) & & F(X) \otimes' F(\top) & \xrightarrow{\mu_{X,\top}} & F(X \otimes \top) & & F(X \otimes Y) & \xrightarrow{F(\sigma)} & F(Y \otimes X)
 \end{array}$$

The first diagram states that the functor must preserve associativity, the second and third that it preserves the unit, and the last that it preserves symmetry. These definitions are taken from Benton's original paper on LNL [7].

Summary of adjoint models The idea behind LNL and adjoint models is roughly that unrestricted computations can be mapped into objects and morphisms in the Cartesian closed half of the model, whereas linear computations will be represented by objects and morphisms in the symmetric monoidal closed half of the model. The symmetric monoidal adjunction between the two categories provides a way to move back and forth between the two classes of computation.

Furthermore, as we noted earlier adjunctions always form monads and comonads. Consequently, this means that the Cartesian half of the model is equipped with a monad, which can be used like the lax modality, and the symmetric monoidal closed half of the model is equipped with a comonad, akin to the bang modality in the linear λ -calculus. However, there is one significant

Intuitionistic types	$\tau ::= 1 \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid G\sigma$
Linear types	$\sigma ::= \top \mid \sigma_1 \otimes \sigma_2 \mid \sigma_1 \multimap \sigma_2 \mid F\tau$
Intuitionistic terms	$e ::= \langle \rangle \mid x \mid \lambda x : \tau . e \mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \mathbf{fst} e \mid \mathbf{snd} e \mid Gm \mid \dots$
Linear terms	$m ::= * \mid \mathbf{let} * = m_1 \mathbf{in} m_2 \mid a \mid \lambda a : \sigma . m \mid m_1 m_2$ $\mid \langle m_1, m_2 \rangle \mid \mathbf{let} \langle a, b \rangle = m_1 \mathbf{in} m_2 \mid Fe \mid G^{-1}e$ $\mid \mathbf{let} Fx = m_1 \mathbf{in} m_2 \mid \dots$
Cartesian term context	$\Gamma ::= \cdot \mid \Gamma, x : \tau$
Linear term context	$\Delta ::= \cdot \mid \Delta, a : \sigma$

Figure 19: The adjunction calculus grammar.

weakness in this model. Because it uses symmetric monoidal closed categories and symmetric monoidal adjunctions, the model only works for monads that are commutative. This does not mean that the model is without value, but there are still important families of monads that are not commutative. For example, the list monad that we presented earlier is not commutative.

2.6.2 The adjoint calculus

In their paper, Benton and Wadler present a term model of LNL that they call the adjoint calculus. This provides a much more convenient target than a purely categorical model for the encodings of λ_{\circ} and ILL that will follow. The grammar of the adjoint calculus is described in Figure 19. The Cartesian half of the model is represented through what are called the Intuitionistic terms and types and the symmetric monoidal closed half through the linear terms and types. Not surprisingly, the terms and types for each half very closely resemble the simply typed λ -calculus and the linear λ -calculus.

The only unusual bits are the type constructors G and F which correspond to the functors F and G in the symmetric monoidal adjunction of the model. The introduction and elimination forms for F are very much like those for **box** in $\lambda_{\square, \diamond}$, which is understandable because they both have similar comonadic structure. The introduction form for G is very much like **val** from λ_{\circ} , but the elimination form does not really look like the join, extension, or bind we might expect for a monadic connective.

The static semantics for the adjoint calculus can be found in Figure 20. Here we write $\Gamma \vdash_{\mathcal{C}} e : \tau$ to mean »Intuitionistic term e has type τ with respect to context Γ « and $\Gamma; \Delta \vdash_{\mathcal{L}} m : \sigma$ to mean »linear term m has type σ with respect to the unrestricted context Γ and the linear context Δ «. The separation of restricted and unrestricted assumptions into separate zones is actually another common way of formulating the linear λ -calculus with the bang modality. In general, the rules are behave much like their simply-typed λ -calculus and linear λ -calculus analogs.

While we do not discuss it here, as it is mostly orthogonal to our points of interest, it is worthwhile to note that adjoint models and the adjoint calculus can be extended to handle coproducts and recursion.

$$\begin{array}{c}
\frac{x:\tau \in \Gamma}{\Gamma \vdash_e x:\tau} \text{wfc:var} \qquad \frac{}{\Gamma; \cdot, a:\sigma \vdash_{\mathcal{L}} a:\sigma} \text{wfl:var} \qquad \frac{}{\Gamma \vdash_e \langle \rangle : 1} \text{wfc:unit} \qquad \frac{}{\Gamma; \cdot \vdash_{\mathcal{L}} *:\top} \text{wfl:unit} \\
\\
\frac{\Gamma; \Delta_1 \vdash_{\mathcal{L}} m_1:\top \quad \Gamma; \Delta_2 \vdash_{\mathcal{L}} m_2:\sigma}{\Gamma; \Delta_1, \Delta_2 \vdash_{\mathcal{L}} \mathbf{let} * = m_1 \mathbf{in} m_2:\sigma} \text{wfl:unite} \qquad \frac{\Gamma \vdash_e e_1:\tau_1 \quad \Gamma \vdash_e e_2:\tau_2}{\Gamma \vdash_e \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{wfc:pair} \\
\\
\frac{\Gamma; \Delta_1 \vdash_{\mathcal{L}} m_1:\sigma_1 \quad \Gamma; \Delta_2 \vdash_{\mathcal{L}} m_2:\sigma_2}{\Gamma; \Delta_1, \Delta_2 \vdash_{\mathcal{L}} \langle m_1, m_2 \rangle : \sigma_1 \otimes \sigma_2} \text{wfl:pair} \qquad \frac{\Gamma \vdash_e e:\tau_1 \times \tau_2}{\Gamma \vdash_e \mathbf{fst} e:\tau_1} \text{wfc:fst} \\
\\
\frac{\Gamma \vdash_e e:\tau_1 \times \tau_2}{\Gamma \vdash_e \mathbf{snd} e:\tau_2} \text{wfc:snd} \qquad \frac{\Gamma; \Delta_1 \vdash_{\mathcal{L}} m_1:\sigma_1 \otimes \sigma_2 \quad \Gamma; \Delta_2, a:\sigma_1, b:\sigma_2 \vdash_{\mathcal{L}} m_2:\sigma}{\Gamma; \Delta_1, \Delta_2 \vdash_{\mathcal{L}} \mathbf{let} \langle a, b \rangle = m_1 \mathbf{in} m_2:\sigma} \text{wfl:proj} \\
\\
\frac{\Gamma, x:\tau_1 \vdash_e e:\tau_2}{\Gamma \vdash_e \lambda x:\tau_1. e:\tau_1 \rightarrow \tau_2} \text{wfc:abs} \qquad \frac{\Gamma; \Delta, a:\sigma_1 \vdash_{\mathcal{L}} m:\sigma_2}{\Gamma; \Delta \vdash_{\mathcal{L}} \lambda x:\sigma_1. m:\sigma_1 \multimap \sigma_2} \text{wfl:abs} \\
\\
\frac{\Gamma \vdash_e e_1:\tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_e e_2:\tau_1}{\Gamma \vdash_e e_1 e_2:\tau_2} \text{wfc:app} \\
\\
\frac{\Gamma; \Delta_1 \vdash_{\mathcal{L}} m_1:\sigma_1 \multimap \sigma_2 \quad \Gamma; \Delta_2 \vdash_{\mathcal{L}} m_2:\sigma_1}{\Gamma; \Delta_1, \Delta_2 \vdash_{\mathcal{L}} m_1 m_2:\sigma_2} \text{wfl:app} \qquad \frac{\Gamma; \cdot \vdash_{\mathcal{L}} m:\sigma}{\Gamma \vdash_e Gm:G\sigma} \text{wfc:G-intro} \\
\\
\frac{\Gamma \vdash_e e:G\sigma}{\Gamma; \cdot \vdash_{\mathcal{L}} G^{-1}e:\sigma} \text{wfl:G-elim} \qquad \frac{\Gamma \vdash_e e:\tau}{\Gamma; \cdot \vdash_{\mathcal{L}} Fe:F\tau} \text{wfl:F-intro} \\
\\
\frac{\Gamma; \Delta_1 \vdash_{\mathcal{L}} m_1:F\tau \quad \Gamma, x:\tau; \Delta_2 \vdash_{\mathcal{L}} m_2:\sigma}{\Gamma; \Delta_1, \Delta_2 \vdash_{\mathcal{L}} \mathbf{let} Fx = m_1 \mathbf{in} m_2:\sigma} \text{wfl:F-elim}
\end{array}$$

Figure 20: The adjunction calculus static semantics.

$$\begin{aligned}
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket &\triangleq \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket \\
\llbracket \circ \tau \rrbracket &\triangleq GF \llbracket \tau \rrbracket \\
\llbracket x \rrbracket &\triangleq x \\
\llbracket \lambda x : \tau . e \rrbracket &\triangleq \lambda x : \llbracket \tau \rrbracket . \llbracket e \rrbracket \\
\llbracket e_1 e_2 \rrbracket &\triangleq \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\
\llbracket \mathbf{val} E \rrbracket &\triangleq G \llbracket E \rrbracket \\
\llbracket [e] \rrbracket &\triangleq F \llbracket e \rrbracket \\
\llbracket \mathbf{let val} x = e \mathbf{in} E \rrbracket &\triangleq \mathbf{let} Fx = G^{-1} \llbracket e \rrbracket \mathbf{in} \llbracket E \rrbracket
\end{aligned}$$

Figure 21: Encoding λ_{\circ} into the adjoint calculus

2.6.3 Encoding λ_{\circ} and ILL into the adjoint calculus

As we mentioned earlier it is possible to encode λ_{\circ} and ILL into the adjoint calculus; Figures 21 and 22 give the translations for λ_{\circ} and ILL respectively based upon those given by Benton and Wadler [9].

The most interesting part of the encodings is the duality between the encoding of \circ and $!$, they become GF and FG respectively²³. Expressions in λ_{\circ} are encoded as linear terms in the adjoint calculus, with G serving as the injection in the the language of terms, and the co-bind for F and G^{-1} handling the threading. The operations for manipulating unrestricted assumptions in ILL all become minor variations on the co-bind of F in the adjoint calculus. This is partly because in accounts of linearity where assumptions are divided into restricted and unrestricted zones, the explicit proof terms necessary for the implementing the structural rules again become implicit.

3 Conclusion

This examination has brought to light a number of interesting open problems and questions. Firstly there appears to be a duality between effect systems and dependency systems that has yet to be explored. It is already well known that confidentiality and integrity are duals, so is there a direct connection between integrity and effects? I conjecture that there is not as integrity is still a dependency property, but it should be examined.

It is also natural to wonder whether there might be a relationship between the monoidal structure of our the label structures that we use to represent effects and dependencies and monoidal structures of monads and comonads. I conjecture that this is just a coincidence.

Except for a brief discussion in Section 2.1, the concept of regions was mostly elided. However, if there does prove to be a duality between effect and dependency type systems, is there a structure that exists on the dependency side of the duality that corresponds to regions? Furthermore, it

²³It is an interesting question as to how this decomposition of the lax modality relates to the decomposition shown by Pfenning and Davies into necessity and possibility [38]

$$\begin{aligned}
\llbracket \top \rrbracket &\triangleq \top \\
\llbracket \tau_1 \multimap \tau_2 \rrbracket &\triangleq \llbracket \tau_1 \rrbracket \multimap \llbracket \tau_2 \rrbracket \\
\llbracket \tau_1 \otimes \tau_2 \rrbracket &\triangleq \llbracket \tau_1 \rrbracket \otimes \llbracket \tau_2 \rrbracket \\
\llbracket !\tau \rrbracket &\triangleq \text{FG}[\llbracket \tau \rrbracket] \\
\llbracket * \rrbracket &\triangleq * \\
\llbracket \text{let } * = e_1 \text{ in } e_2 \rrbracket &\triangleq \text{let } * = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket \\
\llbracket x \rrbracket &\triangleq a \\
\llbracket \text{fn } x : \tau . e \rrbracket &\triangleq \lambda a : \llbracket \tau \rrbracket . \llbracket e \rrbracket \\
\llbracket e_1 e_2 \rrbracket &\triangleq \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\
\llbracket \langle e_1, e_2 \rangle \rrbracket &\triangleq \langle \llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket \rangle \\
\llbracket \text{let } \langle x, y \rangle = e_1 \text{ in } e_2 \rrbracket &\triangleq \text{let } \langle a, b \rangle = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket \\
\llbracket \text{derelict } e \rrbracket &\triangleq \text{let } Fx = \llbracket e \rrbracket \text{ in } G^{-1}x \\
\llbracket \text{discard } e_1 \text{ in } e_2 \rrbracket &\triangleq \text{let } Fx = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket \\
\llbracket \text{copy } e_1 \text{ as } x, y \text{ in } e_2 \rrbracket &\triangleq \text{let } Fx = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket [Fx/a, Fx/b] \\
\llbracket \text{promote } e_1, \dots, e_n \text{ for } x_1, \dots, x_n \text{ in } e \rrbracket &\triangleq \text{let } Fx_1 \dots Fx_n = \llbracket e_1 \rrbracket \dots \llbracket e_n \rrbracket \text{ in } \text{FG}[\llbracket e_2 \rrbracket][Fx_i/a_i]
\end{aligned}$$

Figure 22: Encoding ILL into the adjoint calculus

would be worth investigating whether there is a connection between regions and the nominal calculus used by Nanevski.

There is also the interesting asymmetry in the encoding of λ_{FX} and λ_{SEC} into λ_{O} . In order to be sound, λ_{SEC} must make use of the second upper-bound label, while λ_{FX} does not need to do so. How does this relate to their apparently duality? Does this mean there is something missing from traditional effect type systems?

There is also the problem with the correspondences between ILL, λ_{O} and LNL. It has been noted by Benton and Wadler that ILL and λ_{O} are delightfully close to duals [9]. However as was noted, the encoding into LNL presented only applies to monadic structures that are commutative. Pereira has conjectured [35] that their may be a better fit in attempting to relate monadic languages with ordered substructural logics such as the Lambek calculus [25] and Polakow's ordered linear logic [41].

It would also be worth examining whether adjoint models can be extended to handle the lattice of monads used in the labeled version of λ_{O} . Additionally, given that the lax modality can be decomposed into necessity and possibility, does this have implications for the encoding of λ_{O} in the adjoint calculus?

There is also an interesting relationship between the lax modality and the possibility modality. If possibility has the same monadic structure, why can the lax modality defined in terms of necessity and possibility? I conjecture that possibility might fail to actually form a strong monad

Lastly, it does not appear that there has been much further work since Benton and Wadler's paper on using the adjoint calculus as a basis for a practical language combining linearity and a monadic effect discipline. Given what could be learned from some of the open questions above, we think that it is worth continuing research in practical programming with the adjoint calculus.

Acknowledgements

Many thanks go to Benjamin C. Pierce for guiding me in the direction of this WPE-II topic, and lending me some reference material. Additionally, I appreciate Val Tannen and Jean Gallier volunteering to serve on my WPE-II committee. Steve Zdancewic was quite helpful in pointing me in the right direction with respect to a couple topics in dependency and category theory. Many thanks to Jason Reed for helping me to see the obvious differences between Cartesian Closed Categories and symmetric monoidal closed categories, as well as helping me better understand adjunctions. I am also thankful for my advisor Stephanie Weirich.

References

- [1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, Jan. 1999.
- [2] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.
- [3] S. Abramsky and G. McCusker. Game semantics. In *Proceedings of Marktorberdorf '97 Summer-school, 1997*. Lecture notes.
- [4] L. Augustsson. Cayenne—a language with dependent types. In *Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, Sept. 1998.
- [5] J. Backus. The history of Fortran I, II, and III. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.
- [6] M. Barr and C. Wells. *Category Theory for Computing Science (Third Edition)*. Centre de recherches mathématiques, 1999.
- [7] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. Technical report, University of Cambridge, October 1994. Computer Laboratory Technical Report UCAM-CL-TR-352.
- [8] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proceedings Intl. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 75–90. Springer-Verlag, Berlin, 1993.
- [9] N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science, Brunswick, New Jersey*. IEEE Press, July 1996.
- [10] B.-Y. E. Chang, K. Chaudhuri, , and F. Pfenning. A judgmental analysis of linear logic. Technical report, Carnegie Mellon University, April 2003. Technical Report CMU-CS-03-131R.

- [11] C. Consel. Binding time analysis for high order untyped functional languages. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 264–272. ACM Press, 1990.
- [12] K. Cray, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. Technical report, Carnegie Mellon University, September 2003. Technical Report CMU-CS-03-164.
- [13] R. Davies. A temporal-logic approach to binding-time analysis. In *Logic in Computer Science*, pages 184–195, 1996.
- [14] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69. ACM Press, 2001.
- [15] M. V. H. Fairtlough and M. Mendler. Propositional Lax Logic. *Information and Computation*, 137(1):1–33, August 1997.
- [16] A. Filinski. Representing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457. ACM Press, 1994.
- [17] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *14th Symposium on Logic in Computer Science*, pages 214–224, 1999.
- [18] D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.
- [19] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
- [20] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [21] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 130–141, 1995.
- [22] M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.
- [23] R. Kieburtz. Codata and comonads in haskell. Available as a draft from `ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/haskell199.ps`, July 1999.
- [24] S. A. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.
- [25] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.
- [26] B. Lawvere and S. Schanuel. *Conceptual Mathematics: a First Introduction to Categories*. Cambridge University Press, 1997.

- [27] I. Mackie. Lilac — a functional programming language based on linear logic. Master’s thesis, Imperial College, London, 1991.
- [28] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP ’03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225. ACM Press, 2003.
- [29] A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In *Proceedings of the Conference on Logic of Programs*, pages 219–224. Springer-Verlag, 1985.
- [30] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996.
- [31] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
- [32] A. Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, Carnegie Mellon University, August 2004. Carnegie Mellon Technical Report CMU-CS-04-151.
- [33] A. Pardo. Towards merging recursion and comonads. Technical report, Utrecht University, 2000. Technical Report UU-CS-2000-19.
- [34] S. Park and R. Harper. A logical view of effects, 2004. Available as a draft from <http://www-2.cs.cmu.edu/~rwh/papers/modaleff/short.pdf>.
- [35] F. C. N. Pereira. Personal communication, Feb. 2005.
- [36] S. L. Peyton Jones, R. J. M. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. P. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. L. Wadler. Report on the programming language Haskell 98. <http://haskell.org>, Feb. 1999.
- [37] F. Pfenning. Personal communication, Jan. 2002.
- [38] F. Pfenning and R. Davies. A judgemental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.
- [39] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
- [40] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [41] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001. Carnegie Mellon Technical Report CMU-CS-04-151.
- [42] J. C. Reed. Personal communication, Feb. 2005.
- [43] D. N. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, 1999.
- [44] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, Sept. 1989. Imperial College, London.
- [45] P. Wadler. The marriage of effects and monads. *SIGPLAN Notices*, 34(1):63–74, 1999.

- [46] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
- [47] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.
- [48] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.