November 2005

Cause and Effect: Type Systems for Effects and Dependencies

Geoffrey Washburn
University of Pennsylvania, geoffw@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-05-05.

Cause and Effect: Type Systems for Effects and Dependencies

Abstract

Type systems commonly used in practice today fail to capture essential aspects of program behavior: The effects and dependencies of the programs. In this paper, we examine a prototypical effect type system in the style of Gifford et al., and a canonical example of a dependency type system based upon the work of Zdancewic. Finally, we show how these two type systems can be embedded in a more general framework, a monadic type system as developed by Pfenning and Davies.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-05-05.

Cause and Effect: Type Systems for Effects and Dependencies

Geoffrey Washburn
geoffw@cis.upenn.edu
Technical Report MS-CIS-05-05 (Revision NO 1262)
Department of Computer and Information Science
University of Pennsylvania

November 28th, 2005

Abstract

Type systems commonly used in practice today fail to capture essential aspects of program behavior: The effects and dependencies of the programs. In this paper, we examine a prototypical effect type system in the style of Gifford et al. and a canonical example of a dependency type system based upon the work of Zdancewic. Finally, we show how these two type systems can be embedded in a more general framework, a monadic type system as developed by Pfenning and Davies.

1 Introduction

The ability to precisely reason about programs is important both for automated tools and for programmers. For example, understanding the behavior of a program allows a compiler to perform optimizations that might not otherwise be possible. Likewise, descriptive signatures for application programming interfaces, or other library routines, give programmers a concise understanding of how these functions will interact with the code they themselves write. Just as importantly, if the programmer's tools can automatically provide feedback about the behavior of their code, the programmer will have greater confidence that their code meets their informal specifications.

Dating back as far as the Fortran [Bac81] language in the 1950s, type specifications have guided 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 the output it produces. At a minimum this is a vague form of documentation, allows a compiler to produce the correct invocation and return handlers, and yields a simple method to catch errors caused by feeding a function nonsensical input. Recently, parametric and bounded polymorphism, sometimes called "generics", have become part of mainstream languages[LSAS77, Rey83, IPW01, ECM02]. Polymorphic types document that a function can be reused on any input without changing the behavior, or in the case of bounded polymorphism, that the function can operate on any input with the required interface. For some languages with polymorphism, it is even possible to derive properties and equational laws about functions solely from their type specifications [Rey83, Wad89, PAC94].

However, these type systems still fail to capture essential program behaviors. Realistic programs are more than just collections of idealized mathematical functions composed together: They print information to the screen, write data to memory or external storage, communicate with other programs on the same computer or over a network, or simply fail due to an error condition. Mainstream languages today are not equipped with type systems that can effectively describe and document these sorts of "effectful" behaviors.

1.1 Effects

> effect (ɛˈfɛkt) 1.a. Something accomplished, caused, or produced; a result, consequence. Correlative with cause. [. . .] 3.a. An outward manifestation, sign, token, symptom; an appearance, phenomenon. Obs. [. . .] 4.a. Something which is attained or acquired by an action. Obs.
>
> The Oxford English Dictionary Online

Before we examine type systems that capture the "effectful" behaviors of functions and programs, it is worthwhile to review just what is meant by an "effect". Much like the dictionary definition above describes, an effect can be very informally described as some observable manifestation of the program's execution. Typical examples of effects are allocation and mutation in the heap, drawing to the screen, divergence and non-local control transfers. Unfortunately, in the literature examined as part of this survey, there were no attempts to give a general theory or taxonomy of effects. We conjecture that it would actually be difficult to provide such an account of effects, because the notion of possible effects is very tightly coupled with the machine model used to execute a program. However, in our survey we left that there are two technical accounts that provide a promising start towards a complete classification of effects.

Park and Harper [PH04] split effects into *control effects* and *world effects*. A control effect is a change in the state of the machine executing the program that cannot be explained by a finite number of machine transitions. For example, in a functional language where the machine model is a small-step semantics, raising an exception or calling a continuation would be a control effect. Their notion of a world effect is an action that transitions the program from one "world" to another, much like a Kripke-style possible worlds interpretation of modal logic [Kri63]. Examples of world effects are program behaviors like writing or reading from mutable storage.

Game semantics provides another classification effects that is strikingly similar to the one devised by Park and Harper [AM97]. Briefly, game semantics provides an interpretation for typed programs as a game where a *player* and *opponent* ask each other questions. The simplest game is the game of type integer, where the player asks the opponent for an integer and they respond with an integer. An instance of the game would look like the following:

Player	?
Opponent	42

The game for the type of integer to integer functions involves a reversal in roles, capturing the contravariance of function spaces. The player starts by asking a question,

but instead of answering, the opponent responds by asking the player as question. After the player answers, the opponent will answer the original question. An instance of the game would look like the following:

Player	?	
	Opponent	?
	Player	21
Opponent	42	

This game actually models the space of all integer to integer functions. Each function corresponds to a strategy for playing the game. So the game played above could be the strategy for the function $\lambda x{:}\mathsf{int}.x + x$.

Now that the fundamentals have been explained, we can explain two classes of strategies that correspond to effects. First, is the notion of a *bracketed* strategy. Bracketed strategies are ones where the interaction between player and opponent follows a strictly nested structure, like in the game above. The following game is not bracketed:

Player	?	
	Opponent	?
	Player	21
Player	?	
	Opponent	?
	Player	42
Opponent	42	

This strategy could correspond to a program that uses a captured continuation to return to the start of execution.

Conjecture 1.1. *Strategies that are not bracketed correspond exactly to programs that will exhibit a control effect.*

A second useful category of strategy are the *innocent* strategies. Innocent strategies must behave the same regardless of context. In the following instance, the same game is played twice consecutively:

Player	?
Opponent	21
Player	?
Opponent	42

In this case the opponent's strategy is not innocent, otherwise it would have given the same answer every time it was queried.

Conjecture 1.2. *Strategies that are not innocent correspond exactly to those programs that will exhibit a world effect.*

The similarity between these two accounts of effects is pleasing, but it is not clear whether either of these accounts are complete in the sense that there are no behaviors we would call effects that they do not capture.

1.2 Dependencies

> dependency (dɪˈpɛndənsɪ) 1. The condition of being dependent; the relation of a thing to that by which it is conditioned; contingent logical or causal connexion;
>
> The Oxford English Dictionary Online

Where we informally described an effect as "some observable manifestation of the program's execution," we will take a similar cue from the dictionary definition of dependency and say that a dependency is a logical connection with an observable manifestation of the program's execution. Consequently, we believe the notion of program *dependencies* to be tantalizingly close to the dual of effects.

The distinction is subtle. For example, if a program writes to a memory cell and then returns the integer zero we could give the observable manifestation of that write a name, say $\mathcal{W}$. The fact that the program produced $\mathcal{W}$ is an effect. If the program returned the integer zero, rather than some other integer, *because* $\mathcal{W}$ occurred, then $\mathcal{W}$ is a dependency of that integer.

Knowing about the dependencies of a program or function is just as important for a compiler or a programmer as knowing the effects it may 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* [Zda02], 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" [Wei84]. 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 [Con90, Dav96]. Liveness analysis in optimizing compilers is another example of a dependency analysis, though it is generally not formalized within a type system.

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 analysis and heuristics for discovering the effect and dependency relationships in programs, rather than formal systems for reasoning about effects and dependencies. Limiting our scope to type systems in particular, there are three paradigms we feel have proven enormously influential.

- Effect type systems: In Section 3 we present λ_{FX}, an idealization of prototypical effect type systems.
- Information-flow type systems: In Section 4 we present an information-flow type system, λ_{SEC}, that captures of the essentials of dependency tracking.
- Monadic type systems: In Section 5 we present $\lambda_{\bigcirc}$, a language capable of formalizing the reasoning about both effects and dependencies through a monadic structure.

Finally, in Section 6 we conclude our survey by briefly noting other approaches to tracking effects and dependencies that we have not discussed and avenues for future exploration.

2 Labels

We want to concentrate on the essential differences between formalizing effects and dependencies, so we have chosen to use represent individual effects and dependencies through a unified syntactic category we will call *labels*. A language with labels is then parameterized by a *label structure*.

Definition 2.1. *A label structure $\langle \mathfrak{L}, \leq, \varnothing, \bowtie \rangle$ consists of:*

- *A partially-ordered set of labels, $\langle \mathfrak{L}, \leq \rangle$.*
- *A distinguished least-element of $\mathfrak{L}$, written as $\varnothing$, called the empty label.*
- *An associative, commutative binary operator, $\bowtie$, on elements of $\mathfrak{L} \times \mathfrak{L}$ that we call label join. The set of labels, $\mathfrak{L}$, must be closed under label join: If ℓ_1 and ℓ_2 are in $\mathfrak{L}$, then so is their join $\ell_1 \bowtie \ell_2$. The empty label must act as a unit for label join: If ℓ in $\mathfrak{L}$ then $\ell \bowtie \varnothing = \ell$ and $\varnothing \bowtie \ell = \ell$.*

Abstractly, a label structure is a commutative monoid over a poset. It would be straightforward to use any commutative monoidal structure $\langle \mathfrak{L}, \varnothing, \bowtie \rangle$ by defining the partial order as $\varnothing \leq \ell$ for all ℓ in $\mathfrak{L}$ and $\ell_1 \leq \ell_1 \bowtie \ell_2$ for all ℓ_1, ℓ_2 in $\mathfrak{L}$.

We call a label *atomic* if it cannot be expressed as the join of two or more other labels in $\mathfrak{L}$.

2.1 Examples

As an aid to understanding how label structures are used to represent effects and dependencies, we describe two example label structures.

Regions

A common application of effect systems is to keep track of how a program will interact with heap allocated memory. To simplify the analysis, the focus is often restricted to interactions with *regions* of the heap rather than individual memory locations. Regions evolved out of the work by Talpin and Jourvelot [TJ92] on effect type systems and memory management. We use the metavariable ρ to range over regions of the heap.

We can then define two atomic labels, $\mathsf{read}(\rho)$ and $\mathsf{write}(\rho)$, corresponding to the action of reading and writing to a region ρ in the heap. The label set $\mathfrak{L}$ is the free commutative monoid over $\mathsf{read}(\rho)$ and $\mathsf{write}(\rho)$ for all regions ρ. We then order $\mathfrak{L}$ by set inclusion.

The label $\mathsf{write}(\rho_1) \bowtie \mathsf{read}(\rho_2)$ indicates the act of writing to the region ρ_1 and reading from the region ρ_2. Labels as we have defined do not capture the order in which the operations occur. It is possible to design type systems that capture this level of precision, but we do not explore them in this paper.

Security levels

Another concrete label structure is the two-level security lattice commonly used in information-flow type systems [BL75]. A label $\top$ indicates the dependence or use of "high-security" information while $\bot$ indicates the dependence or use of only "low-security" information.

We then take $\mathfrak{L}$ to be the set $\{\bot, \top\}$. We define the partial order on $\mathfrak{L}$ as $\bot \leq \top$, label join as $\ell \bowtie \top = \top \bowtie \ell = \top$, and $\bot$ is the empty label.

3 The λ_{FX} language

The language λ_{FX} is our creation, but is novel only as a distillation of effect type systems as originally developed by Gifford et. al [GJLS87]. The grammar for λ_{FX} can be found

$$\begin{array}{rl} \text{Types} & \tau ::= \mathsf{int} \mid \tau_1 \xrightarrow{\ell} \tau_2 \mid \ldots \\ \text{Terms} & e ::= \mathsf{i} \mid \mathsf{x} \mid \lambda \mathsf{x}{:}\tau.e \mid e_1 e_2 \mid \ldots \\ \text{Term variable context} & \Gamma ::= \cdot \mid \Gamma, \mathsf{x}{:}\tau \end{array}$$

Figure 1: The grammar for λ_{FX}.

$$\frac{}{\Gamma \vdash_{\varnothing} \mathsf{i} : \mathsf{int}}\ \textsc{wft:int} \qquad \frac{\mathsf{x}{:}\tau \in \Gamma}{\Gamma \vdash_{\varnothing} \mathsf{x} : \tau}\ \textsc{wft:var}$$

$$\frac{\Gamma, \mathsf{x}{:}\tau_1 \vdash_{\ell} e : \tau_2}{\Gamma \vdash_{\varnothing} \lambda \mathsf{x}{:}\tau.e : \tau_1 \xrightarrow{\ell} \tau_2}\ \textsc{wft:abs} \qquad \frac{\Gamma \vdash_{\ell_1} e_1 : \tau_1 \xrightarrow{\ell} \tau_2 \qquad \Gamma \vdash_{\ell_2} e_2 : \tau_1}{\Gamma \vdash_{\ell_1 \bowtie \ell_2 \bowtie \ell} e_1 e_2 : \tau_2}\ \textsc{wft:app}$$

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

Figure 2: The static semantics for λ_{FX}.

in Figure 1. The only difference from the simply-typed λ-calculus is that function types in λ_{FX} are annotated with a label. For now, we do not assume any concrete label structure. We write the type $\tau_1 \xrightarrow{\ell} \tau_2$ to mean the type of a function from values of type τ_1 to values of type τ_2 that may cause some effect ℓ as a result of evaluation. This effect is sometimes called the *latent* effect of the function.

The static semantics for λ_{FX} can be found in Figure 2. We write the judgment $\Gamma \vdash_{\ell} e : \tau$ to mean "term e has type τ and may produce an effect ℓ with respect to the context Γ." The key difference between judgments in λ_{FX} and in the simply-typed λ-calculus is that we decorate the turnstiles of the judgments with a label. This label is intended to provide a conservative estimate of the effect that will be produced when evaluating the term. We sometimes describe terms where the concluding judgment is the empty label as *inert*.

Because λ-abstractions suspend the evaluation of their bodies, the rule WFT:ABS correspondingly captures the effect that may be produced by the body and records it in the function type. Abstractions are themselves values, meaning that they can take no reduction steps. Because they have no reductions, there is no opportunity for them to cause an effect. Consequently, we deem them inert and indicate this by the empty label in the conclusion. Similarly, integer values are considered inert in the rule WFT:INT.

$$\frac{}{\tau \leq \tau}\ \textsc{sub:refl} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \textsc{sub:trans}$$

$$\frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4 \qquad \ell_1 \leq \ell_2}{(\tau_1 \xrightarrow{\ell_1} \tau_2) \leq (\tau_3 \xrightarrow{\ell_2} \tau_4)}\ \textsc{sub:arr}$$

Figure 3: The subtyping relation for λ_{FX}.

Intuitively, when a function application is evaluated the two terms will produce some effects, this is formalized in the premises of the WFT:APP rule. Additionally, the actual evaluation of the resulting function will release its latent effect, as indicated by the effect label decoration in the conclusion of the the WFT:APP rule.

The typing rule WFT:SUB provides for subsumption of the effect annotation and types. Because labels have a partial order and occur in types, this partial order induces a standard subtyping relationship defined in Figure 3.

It is important to note that the static semantics we present for λ_{FX} in Figure 2 is only sound for a call-by-value operational semantics. This is because the typing rule for variables, WFT:VAR, assumes that only inert values will be substituted for variables. Therefore, rather than the most general substitution theorem we might expect, only the following weaker theorem holds

Theorem 3.1 (Substitution)**.** *If* $\Gamma \vdash_{\varnothing} e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1 \vdash_{\ell} e_2 : \tau_2$ *then* $\Gamma \vdash_{\ell} e_2[e_1/x] : \tau_2$.

Proof. By straightforward induction over the structure of $\Gamma, x{:}\tau_1 \vdash_{\ell} e_2 : \tau_2$. □

3.1 Examples

We now sketch a few examples to illustrate how λ_{FX} can be used to model various kinds of effectful operations.

Region based memory management

One natural extension of λ_{FX} is region-based memory management. We extend the language with a new type constructor for reference cells, ref τ, which is used to indicate a value of type τ stored in region ρ of the heap. Additionally, we introduce four new term forms: Region allocation ($\nu\rho.e$), reference cell allocation (**ref** e), dereferencing

$$\frac{\Gamma \vdash_{\ell} e : \tau \qquad \rho \notin \tau}{\Gamma \vdash_{\ell -} \nu\rho.e : \tau}\ \text{WFT:NU} \qquad \frac{\Gamma \vdash_{\ell} e : \tau}{\Gamma \vdash_{\ell \bowtie \mathsf{new}(\ )} \mathbf{ref}\ \ e : \mathsf{ref}\ \ \tau}\ \text{WFT:REF}$$

$$\frac{\Gamma \vdash_{\ell} e : \mathsf{ref}\ \ \tau}{\Gamma \vdash_{\ell \bowtie \mathsf{read}(\ )} !e : \tau}\ \text{WFT:DEREF} \qquad \frac{\Gamma \vdash_{\ell} e_1 : \mathsf{ref}\ \ \tau \qquad \Gamma \vdash_{\ell} e_2 : \tau}{\Gamma \vdash_{\ell \bowtie \mathsf{write}(\ )} e_1 := e_2 : \mathsf{ref}\ \ \tau}\ \text{WFT:ASSN}$$

Figure 4: Extensions to the λ_{FX} type system for regions and reference cells.

($!e$), and assignment ($e_1 := e_2$). The static semantics of these new terms is shown in Figure 4. This extension does not change the subtyping relation, but it is important to note that this implies that reference cell types are invariant under subtyping.

Next we need to define the label structure for this extension to λ_{FX}. The label structure is essentially the same as we described for regions in Section 2, but with a third atomic label $\mathsf{new}(\rho)$. The term **ref** e allocates a new reference cell in region ρ, initializing it with the result of evaluating e. This propagates an effect label $\mathsf{new}(\rho)$ indicating that evaluating the term will allocate a reference cell in region ρ. Similarly, the rules for dereferencing a cell and assigning to a cell propagate effects $\mathsf{read}(\rho)$ and $\mathsf{write}(\rho)$ respectively.

The typing rule for region allocation is the most interesting. Operationally we would like the region allocation term to evaluate in the following manner:

1. allocate a new region for ρ;
2. evaluate its body, e;
3. deallocate the region ρ;
4. return the value produced by evaluating its body.

In order for this sequence of operations to be sound, we must ensure that, after ρ has been deallocated, there will be no further references to it. In λ_{FX}, the only way that this could occur is if ρ were captured in the closure of a λ-abstraction and then used later. However, if this were the case, ρ would show up in the latent effect of the function. This is the motivation behind the precondition $\rho \notin \tau$ in WFT:NU.

The other interesting aspect of region allocation is that can also "mask" or eliminate effects involving the allocated region. This not only helps eliminate spurious effects appearing in types and the effect decoration, but is necessary because it would mean that ρ would escape its scope via labels. Therefore, in the conclusion of the rule we

write $\ell - \rho$. This is shorthand for the finding the largest ρ-free ℓ_1 such that there exists some ℓ_2 where their join, $\ell_1 \bowtie \ell_2$, is equal to ℓ.

With these extensions to λ_{FX} we can now write and type terms like the following

$$\cdot \vdash_{\varnothing} \lambda x{:}\mathsf{int}.(\lambda y{:}\mathsf{ref}\ \ \mathsf{int}.0)(\mathbf{ref}\ \ x) : \mathsf{int} \xrightarrow{\mathsf{new}(\ )} \mathsf{int} \tag{1}$$

This function takes an integer and allocates a reference cell in region ρ using the integer argument as its initial contents. 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, $\mathsf{int} \xrightarrow{\mathsf{new}(\ )} \mathsf{int}$. This says that every time this function is invoked it allocate a new reference cell in region ρ.

To see how the region effect discipline can be used to implement static memory reclamation, consider the implementation of a hypothetical swap function. For simplicity, we assume additional extensions for **let** expressions, sequencing expressions, and unit values.

$$\begin{array}{l} \cdot \vdash_{\varnothing} \lambda x : \mathsf{ref}\ \ {}^{1}\ \mathsf{int}.\lambda y : \mathsf{ref}\ \ {}^{2}\ \mathsf{int}. \\ \qquad \nu\rho.\mathbf{let}\ z{:}(\mathsf{ref}\ \ \mathsf{int}) = (\mathbf{ref}\ \ !x)\ \mathbf{in} \\ \qquad\quad (x := (!y); y := (!z); \langle\rangle) \\ \quad : \mathsf{ref}\ \ {}^{1}\ \mathsf{int} \xrightarrow{\varnothing} \mathsf{ref}\ \ {}^{2}\ \mathsf{int} \xrightarrow{\mathsf{read}(\ {}_1) \bowtie \mathsf{read}(\ {}_2) \bowtie \mathsf{write}(\ {}_1) \bowtie \mathsf{write}(\ {}_2)} 1 \end{array} \tag{2}$$

This function takes two reference cells, x and y, containing integers and swaps the contents of cells. To do so, a new region ρ is allocated to store a new reference cell, z, to be used as a temporary while swapping. Because there no references to the region ρ outside of the scope of the ν expression, as is enforced by the typing rule WFT:NU, the compiler may generate code to free the reference cell allocated region ρ when the expression has finished evaluating.

Region based memory management is just one example of how an effect system allows us to statically describe effectful behavior.

Nontermination

Another property of programs we might want to track statically is the control effect caused by nontermination. Being able to track non-termination is useful in a number of contexts. If a compiler knows that a bit of code is guaranteed to terminate it can perform optimizations that would otherwise not be possible. In languages with rich dependent type systems, it is important that it is possible to decide type equality. If only terminating terms may appear in dependent types, the problem becomes significantly more tractable.

Because λ_{FX} is only sound for call-by-value semantics, we will define general recursion here using recursive function definitions (**fun** $f(x{:}\tau_1){:}\tau_2.e$) rather than a simpler fixed-point operator.

$$\frac{\Gamma, f{:}\tau_1 \xrightarrow{\ell} \tau_2, x{:}\tau_1 \vdash_\ell e : \tau_2 \qquad \mathsf{div} \leq \ell}{\Gamma \vdash_\varnothing \textbf{fun}\ f(x{:}\tau_1){:}\tau_2.e : \tau_1 \xrightarrow{\ell} \tau_2}\ \textsc{wft:fun}$$

We define the label structure for this extension to be the free commutative-monoid formed from the single atomic label div (short for divergence). The precondition $\mathsf{div} \leq \ell$ in the rule WFT:FUN enforces that the label ℓ must at least indicate the effect of divergence, but could perhaps have additional effects as well.

However, this typing rule is necessarily conservative because determining whether arbitrary general recursive functions terminate is undecidable. Therefore, functions that we intuitively know are terminating, such as factorial (assuming extensions for multiplication, subtraction, booleans, integer comparison, and and conditionals) will be deemed to potentially diverge

$$\cdot \vdash_\varnothing \textbf{fun}\ \mathtt{fact}(x{:}\mathsf{int}){:}\mathsf{int}.\textbf{if}\ x \leq 0\ \textbf{then}\ 1\ \textbf{else}\ x * \mathtt{fact}(x-1) : \mathsf{int} \xrightarrow{\mathsf{div}} \mathsf{int} \qquad (3)$$

One solution would be to add primitive recursive operators to the language so that simple recursive operations will not be incorrectly flagged a possibly diverging.

In addition to the usual type soundness properties of preservation and progress, an added soundness property of this extension to λ_{FX} is that if the concluding judgment of a term's typing derivation does not contain the label div, then it must always terminate. We could express this formally in the following conjectured theorem.

Conjecture 3.2 (Termination soundness)**.** *If* $\cdot \vdash_\ell e : \tau$ *and* div *is not in* ℓ*, then evaluation of* e *always terminates.*

Combining effect systems

Given the two effect systems we have seen, it is natural to desire an effect type system that combines the benefits of region-based memory management and general recursion. Unfortunately, it is not possible to naïvely combine the two. Consider the following program

$$\cdot \vdash_{\substack{\mathsf{new}(\ ) \bowtie \\ \mathsf{write}(\ ) \bowtie \\ \mathsf{read}(\ )}} \begin{array}{l} \textbf{let}\ x{:}\mathsf{ref}\ \ \mathsf{int} \xrightarrow{\mathsf{read}(\ )} \mathsf{int} = \textbf{ref}\ \ (\lambda y{:}\mathsf{int}.0)\ \textbf{in} \\ \quad \textbf{let}\ z{:}\mathsf{ref}\ \ \mathsf{int} \xrightarrow{\mathsf{read}(\ )} \mathsf{int} = (x := (\lambda y{:}\mathsf{int}.(!x)y))\ \textbf{in} \\ \qquad (!z)0 \end{array} : \mathsf{int} \qquad (4)$$

Types	τ	::=	$\mathsf{int}_\ell \mid \tau_1 \to_\ell \tau_2 \mid \ldots$
Terms	e	::=	$i \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \ldots$
Term variable context	Γ	::=	$\cdot \mid \Gamma, x{:}\tau$

Figure 5: The grammar for λ_{SEC}.

While well-typed in a naïve combination of the extensions we have presented for region-based memory management and general recursion, it is not sound. This program will diverge, but the effect label div will not appear anywhere in the typing derivation. This violates the desired termination soundness theorem we proved in the preceding subsection.

It is possible to recover termination soundness by changing the typing rules for region-based memory management to introduce the divergence label, div, this will, however, result in an even more conservative description of a program's termination behavior.

4 The λ_{SEC} language

We present λ_{SEC} as a canonical example of a type system for tracking dependencies. Originally developed by Zdancewic [Zda02] as a core calculus for studying secure information-flow, λ_{SEC} could, with minor modifications, be a suitable basis for statically describing many dependency analysis such as program slicing, call-tracking, or binding-time analysis. The grammar for λ_{SEC} can be found in Figure 5.

In λ_{SEC} integer types are labeled, as are integers. However, unlike like λ_{FX}, these labels are used not to represent effects that a term with that type might produce, but a conservative estimate of their dependencies. The stock example of a dependency analysis is secure information-flow using the two level label lattice we introduced in Section 2. We write $\bot$ for a dependency on low security information and $\top$ for a dependency on high security information. Therefore, the type $\mathsf{int}_\bot$ describes terms whose computation will only depend upon low-security information, while those with type $\mathsf{int}_\top$ may depend upon high-security information. The label on a function type, $\tau_1 \to_\ell \tau_2$, indicates that the computation by which the function itself is created will depend upon information ℓ.

The typing rules for λ_{SEC} can be found in Figure 6. We write the judgment $\Gamma \vdash e : \tau$ to mean that "term e has type τ with respect to term variable context Γ". Because dependencies are captured entirely through a term's type, no additional decoration is on the judgment is needed like in λ_{FX}.

$$\frac{}{\Gamma \vdash i : \mathsf{int}_{\varnothing}}\ \textsc{wft:int} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{wft:var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau.e : \tau_1 \to_{\varnothing} \tau_2}\ \textsc{wft:abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to_{\ell} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2 \bowtie \ell}\ \textsc{wft:app} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{wft:sub}$$

Figure 6: The static semantics for λ_{SEC}.

$$\frac{}{\tau \leq \tau}\ \textsc{sub:refl} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \textsc{sub:trans} \qquad \frac{\ell_1 \leq \ell_2}{\mathsf{int}_{\ell_1} \leq \mathsf{int}_{\ell_2}}\ \textsc{sub:int}$$

$$\frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4 \qquad \ell_1 \leq \ell_2}{(\tau_1 \to_{\ell_1} \tau_2) \leq (\tau_3 \to_{\ell_2} \tau_4)}\ \textsc{sub:arr}$$

Figure 7: The subtyping relation for λ_{SEC}.

As defined in the rules WFT:INT and WFT:ABS, integer and functions values have no dependencies initially because nothing needed to have happened for them to be created. Term variables just receive whatever type is specified by term variable context, as the type completely describes the dependencies of the terms that may be substituted for it. As with, λ_{FX}, the partial order on labels induces a subtyping relation for λ_{SEC} and the rule WFT:SUB can be used to change the type of a term by subsumption. The subtyping relation for λ_{SEC} is defined in Figure 7.

The most interesting typing rule is WFT:APP, for application. Here given a function with type $\tau_1 \to_{\ell} \tau_2$ and an argument of type τ_1 the application will be of type $\tau_2 \bowtie \ell$. We write $\tau \bowtie \ell$ as shorthand for the following:

$$\begin{aligned} \mathsf{int}_{\ell_1} \bowtie \ell_2 &\triangleq \mathsf{int}_{(\ell_1 \bowtie \ell_2)} \\ (\tau_1 \to_{\ell_1} \tau_2) \bowtie \ell_2 &\triangleq \tau_1 \to_{(\ell_1 \bowtie \ell_2)} \tau 2 \end{aligned}$$

The application receives this type because the result necessarily depends upon the function that computes it. Therefore, we must augment the dependencies of τ_2 with the additional dependencies, ℓ, that the function contributes.

Unlike, λ_{FX}, the type system of λ_{SEC} is agnostic to evaluation order. Again, as a consequence of dependencies being captured entirely within types, the most general substitution theorem possible is true.

$$\frac{}{\Gamma \vdash \mathbf{passwd} : \mathsf{int}_\top}\ \textsc{wft:pw} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int}_\ell \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if0}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \bowtie \ell}\ \textsc{wft:ifz}$$

Figure 8: Extensions to λ_{SEC} for secure information-flow

Theorem 4.1 (Substitution). *If* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$ *then* $\Gamma \vdash e_2[e_1/x] : \tau_2$.

Proof. By straightforward induction over the structure of $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$. □

4.1 Examples

Now we examine instantiations of λ_{SEC} for comparison with λ_{FX}.

We will not examine how λ_{SEC} interacts with region-based memory management because it introduces significant complications that are beyond the scope of this paper. Without presenting region-based memory management, we do not have a direct example, but we conjecture that as with λ_{FX}, it is not sound to naïvely compose instantiations of λ_{SEC}.

Secure information-flow

As we have mentioned previously, a common application for λ_{SEC} is as a security-type system. Here the goal is to track what parts of a program will depend upon high security information. This is valuable both prescriptively and descriptively: Type annotations can enforce that inputs and outputs do not make use of sensitive information and type inference allows for the discovery of unexpected places to which sensitive information might flow.

As a simple example, we can consider adding a distinguished integer constant, **passwd**, to λ_{SEC} that corresponds to a password, and a conditional for integers. The typing rules for this extension can be found in Figure 8. We again use the two level label lattice we introduced in Section 2. Unlike the usual integers, the typing rule WFT:PW gives **passwd** a type that is labeled with $\top$.

The typing rule for conditionals is abstractly an elimination form for integers, much like function application is the elimination form for function values. Therefore, just like the typing rule for applications, WFT:IFZ relabels the type of the branches to indicate that result of the computation will inherit the dependencies of the scrutinee. Consider the following trivial example.

$$\cdot \vdash \mathbf{if0}\ \mathbf{passwd}\ \mathbf{then}\ 0\ \mathbf{else}\ 1 : \mathsf{int}_\top \tag{5}$$

Here the type of the entire term is given a "high-security" label, because the choice of whether to return 0 or 1 depends upon high-security information. However, the typing rule is necessarily conservative and will give the same type in the following example.

$$\cdot \vdash \textbf{if0}\ 0\ \textbf{passwd then}\ 0\ \textbf{else}\ 0 : \mathsf{int}_\top \tag{6}$$

Here, the result will be 0 regardless of the value of **passwd**, so it would be sensible to actually give the term the type $\mathsf{int}_\bot$ because the answer does not really depend upon **passwd**. However, in general deciding whether the branches of the conditional will always yield the same result is undecidable. So, erring toward a decidable typechecking algorithm means that we have no choice but to accept this imprecision.

Because **passwd** is the only innately high-security value in the language, we can necessarily conclude that any term that has a type labelled with $\top$ *may have* depended upon its value. However, we can draw the much more definitive conclusion for terms with a type labeled with the empty-label – that they *did not depend* upon the value of **passwd**.

We can make this informal reasoning precise in a theorem called noninterference.

Theorem 4.2 (Noninterference)**.** *If* $\cdot, x : \mathsf{int}_\top \vdash e : \mathsf{int}_\bot$ *then for any two* $\cdot \vdash e_1 : \mathsf{int}_\top$ *and* $\cdot \vdash e_2 : \mathsf{int}_\top$ *the result of computing* $e[e_1/x]$ *and* $e[e_2/x]$ *will be equivalent.*

Proof. The canonical proof is a corollary of a more general proof of substitution for a logical relation, but purely syntactic techniques also exist. □

Nontermination

For comparison with λ_{FX}, we can also examine nontermination using dependency in λ_{SEC}. Because λ_{SEC} is agnostic to evaluation order, we can most easily define general recursion using a fix-point operator.

$$\frac{\Gamma, x{:}\tau \vdash e : \tau \qquad \mathsf{div} \leq \tau}{\Gamma \vdash \textbf{fix}\ x{:}\tau.e : \tau}\ \text{WFT:FIX}$$

This rule looks much like the rule for recursive functions we gave for λ_{FX}. In the rule we have written $\ell \leq \tau$ as short-hand for

$$\begin{array}{rcl} \ell_1 \leq \mathsf{int}_{\ell_2} & \triangleq & \ell_1 \leq \ell_2 \\ \ell_1 \leq (\tau_1 \to_{\ell_2} \tau_2) & \triangleq & \ell_1 \leq \ell_2 \end{array}$$

Instead of requiring that the latent effect on the recursive function have at least the effect of divergence, for λ_{SEC} we simply check that type is labeled as being dependent upon on at least divergence.

As in λ_{FX}, tracking nontermination is conservative, but if we revisit our factorial example, we see that it unexpectedly more conservative.

$$\begin{array}{l}\cdot \vdash \textbf{fix}\ \mathtt{fact} : (\mathsf{int}_{\ell} \rightarrow_{\mathsf{div}} \mathsf{int}_{(\ell \bowtie \mathsf{div})}).\lambda x{:}\mathsf{int}_{\ell}.\ \ : \mathsf{int}_{\ell} \rightarrow_{\mathsf{div}} \mathsf{int}_{(\ell \bowtie \mathsf{div})} \\ \qquad \textbf{if}\ x \leq 0\ \textbf{then}\ 1\ \textbf{else}\ x * \mathtt{fact}(x-1)\end{array} \tag{7}$$

Here the type system has said that the function itself may depend upon a diverging computation, which is not what we might have expected. It is entirely correct however. Instead of defining factorial directly using a recursive function, we have written a program that computes the factorial function using a fix-point. Because we must conservatively assume that the result of fix-point computation may depend upon a diverging computation, the function it computes inherits this dependency.

However, it is straightforward to define recursive functions in λ_{SEC}.

$$\frac{\Gamma, f{:}(\tau_1 \rightarrow_{\varnothing} \tau_2), x{:}\tau_1 \vdash e : \tau_2 \qquad \mathsf{div} \leq \tau_2}{\Gamma \vdash \textbf{fun}\ f(x{:}\tau_1){:}\tau_2.e : \tau_1 \rightarrow_{\varnothing} \tau_2}\ \textsc{wft:fun}$$

This looks even more like the rule from λ_{FX}, except that instead of ensuring that the latent effect of the function is at least that of divergence, we check that the result of computing the body indicates that it could depend upon a diverging computation.

It is now possible to rewrite factorial, obtaining a more suitable type and dependency analysis.

$$\begin{array}{l}\cdot \vdash \textbf{fun}\ \mathtt{fact} : (x{:}\mathsf{int}_{\ell}){:}\mathsf{int}_{(\ell \bowtie \mathsf{div})}. \qquad\qquad : \mathsf{int}_{\ell} \rightarrow_{\varnothing} \mathsf{int}_{(\ell \bowtie \mathsf{div})} \\ \qquad \textbf{if}\ x \leq 0\ \textbf{then}\ 1\ \textbf{else}\ x * \mathtt{fact}(x-1)\end{array} \tag{8}$$

In general, introducing functionality as a primitive, rather than reducing it to syntactic sugar, will allow for more precise reasoning about its dependencies.

Just as we could in λ_{FX}, we could use this machinery to prove additional soundness properties about λ_{SEC}.

Conjecture 4.3 (Termination soundness)**.** *If* $\cdot \vdash e : \tau$ *and* $\mathsf{div} \not\leq \tau$*, then evaluation of* e *always terminates.*

5 The $\lambda_{\bigcirc}$ language

We presented λ_{FX} as an example of how to capture effectual computations in a type system and we presented λ_{SEC} as a prototypical type system for tracking dependency relationships. Interestingly, despite the seemingly dual relationship between effects and dependencies, we can embed them both into a single language, $\lambda_{\bigcirc}$.

We call $\bigcirc$ the "lax" modality because $\lambda_{\bigcirc}$ is the computational interpretation of what is known as "lax logic" [FM97]. Lax logic is in turn actually a logical interpretation of Eugeno Moggi's monadic meta-language [Mog91], which he developed for reasoning about programs with effectful behavior. The lax modality's strong monadic structure has become so pervasively used for encapsulating effects in pure languages, such as Haskell [PHA+99], that it is often simply called a "monad".

5.1 Category theoretic foundations

Before introducing $\lambda_{\bigcirc}$, we will take a brief detour through the category theoretic foundations of the lax modality.

Definition 5.1. *A monad (also called a triple or Kleisli triple) over a category $\mathcal{C}$ is a structure $\langle F, \eta, \star \rangle$ consisting the following components:*

- $F : \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{C})$*, a function from objects of a category $\mathcal{C}$ to themselves.*
- *A family of morphisms $\eta_X : X \to F(X)$, for all objects X in the category $\mathcal{C}$.*
- *An operator $\star$ such that for any morphism $f : X \to F(Y)$ in the category $\mathcal{C}$, where X and Y are objects of $\mathcal{C}$, there exists a morphism $f^\star : F(X) \to F(Y)$ in $\mathcal{C}$.*

Furthermore, a monad must satisfy the following equational laws:

- *For all objects X in the category $\mathcal{C}$, $\eta_X^\star = \mathrm{id}_{F(X)}$.*
- *For all morphisms $f : X \to F(Y)$ in the category $\mathcal{C}$, $f^\star \circ \eta_X = f$.*
- *For all morphisms $f : X \to F(Y)$ and $g : Y \to F(Z)$ in the category $\mathcal{C}$, $g^\star \circ f^\star = (g^\star \circ f)^\star$.*

The morphisms η_X are called the *units* of the monad. The operator $\star$ is often called the *extension*. However, it is often convenient to treat the extension as a binary operator, called *bind*, that takes objects $F(X)$ and morphisms $f : X \to F(Y)$ in $\mathcal{C}$ to objects $F(Y)$ in $\mathcal{C}$. In practical programming, the bind operator turns out to be a more intuitive way of working with extension. So for the remainder of our presentation we treat $\star$ as the binary bind operator.

Returning to the domain of programming languages, it is common to model the simply-typed λ-calculus by a Cartesian Closed Category. Precisely, a Cartesian Closed Category is any category with finite products, terminal objects, and exponentials. Informally, we can think of a Cartesian Closed Category as a category where the objects are types and the morphisms $f : \tau_1 \to \tau_2$ are terms of type τ_2 with a single free variable of type τ_1. Using this model, it straightforward to show that any number of

$$
\begin{array}{rrcl}
\text{Types} & \tau & ::= & \mathsf{int} \mid \tau_1 \to \tau_2 \mid \bigcirc\tau \mid \ldots \\
\text{Terms} & e & ::= & \mathfrak{i} \mid x \mid \lambda x{:}\tau_1.e \mid e_1 e_2 \mid \mathbf{val}\ E \mid \ldots \\
\text{Expressions} & E & ::= & [e] \mid \mathbf{let\ val}\ x = e\ \mathbf{in}\ E \mid \ldots \\
\text{Term variable context} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau
\end{array}
$$

Figure 9: The grammar for $\lambda_{\bigcirc}$.

type constructors and associated operations form a monad. A very simple example is forming a monad from the list type constructor.

$$
\begin{array}{rcl}
F & \triangleq & \mathsf{list} \\
\eta & \triangleq & \lambda x{:}\tau.[x] \\
\star & \triangleq & \lambda x{:}F(\tau_1).\lambda f{:}\tau_1 \to F(\tau_2).\mathtt{flatten}(\mathtt{map}\ f\ x)
\end{array}
$$

Here we take the monad's function F to be the type constructor list. The monad's units are the family of functions from any given type to the singleton list of that type. The monad's bind operator maps its function argument over its list argument, and flattens the result back into a list. As it stands, this monad is rather boring because it only ever computes with singletons lists. However, the structure itself can form the basis for a rich theory of list comprehensions [Wad92].

Lists, however, are rather orthogonal our central theme: effects and dependencies. As suggested by Moggi, it is more useful to think of objects τ as being values of type τ and objects $F(\tau)$ as computations of type τ. This led Moggi to call the function F a *notion of computation*, because it abstracts away from the values used to represent a given sort of computation. For example, mutable state and exceptions may be conveniently abstracted by using the following notions of computation:

- **mutable state**: $F \triangleq \lambda\alpha.\sigma \to (\alpha \times \sigma)$, where σ is the type of the mutable store.
- **exceptions**: $F \triangleq \lambda\alpha.\alpha + \sigma$, where σ is the type of the exception packet.

Here, computations concerning mutable state are represented by values that are functions and computations with exceptions are represented by values that are sums. The function F abstracts away from these concrete representations of computation, letting us reason about computations in the general.

5.2 A judgmental reconstruction

If we wish to abstract away from specific instantiations of a notion of computation, $\lambda_{\bigcirc}$ provides idealized language for distinguishing between values and computations.

$$\frac{}{\Gamma \vdash i : \mathsf{int}}\ \textsc{wft:int} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{wft:var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau.e : \tau_1 \to \tau_2}\ \textsc{wft:abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ \textsc{wft:app} \qquad \frac{\Gamma \vdash e \div \tau}{\Gamma \vdash \mathbf{val}\ e : \bigcirc\tau}\ \textsc{wft:val}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] \div \tau}\ \textsc{wfe:unit} \qquad \frac{\Gamma \vdash e : \bigcirc\tau_1 \qquad \Gamma, x{:}\tau_1 \vdash E \div \tau_2}{\Gamma \vdash \mathbf{let\ val}\ x = e\ \mathbf{in}\ E \div \tau_2}\ \textsc{wfe:let}$$

Figure 10: The static semantics for $\lambda_{\bigcirc}$.

Expression contexts	$\mathcal{E}$	::=	$\bullet \mid [\mathcal{T}] \mid \mathbf{let\ val}\ x = \mathcal{T}\ \mathbf{in}\ E \mid \mathbf{let\ val}\ x = \mathbf{val}\ \mathcal{E}\ \mathbf{in}\ E$
Term contexts	$\mathcal{T}$	::=	$\bullet \mid \mathcal{T} e \mid$

$$\frac{}{(\lambda x{:}\tau.e_1)e_2 \rightsquigarrow\ e_1[e_2/x]}\ \textsc{evb:app}$$

$$\frac{}{\mathbf{let\ val}\ x = \mathbf{val}\ [v]\ \mathbf{in}\ E \rightsquigarrow\ E[v/x]}\ \textsc{evb:letv}$$

$$\frac{E \rightsquigarrow\ E'}{\mathcal{E}\{E\} \rightsquigarrow \mathcal{E}\{E'\}}\ \textsc{ev:ctx1} \qquad \frac{e \rightsquigarrow\ e'}{\mathcal{E}\{e\} \rightsquigarrow \mathcal{E}\{e'\}}\ \textsc{ev:ctx2}$$

Figure 11: The dynamics semantics for $\lambda_{\bigcirc}$.

The grammar is given in Figure 9. Our presentation follows the judgmental precepts of Pfenning and Davies [PD01], and separates the language into "pure" terms and "effectful" expressions. We chose to use this definition because of its clear logical foundation and felt that it made for a less ad-hoc dynamic semantics. Unlike in λ_{FX} and λ_{SEC}, in $\lambda_{\bigcirc}$ a complete program is an expressions rather than a term.

At the level of types, $\lambda_{\bigcirc}$ extends the simply-typed λ-calculus with the modal type constructor $\bigcirc$. This type constructor serves as a generic instance of the monadic function F. At the level of terms, **val** E provides an injection from the language of expressions into the language of terms.

In $\lambda_{\bigcirc}$, computations are represented using a new syntactic category called expressions. The square bracket operator $[\cdot]$ serves as the unit operator of $\bigcirc$ by lifting terms to expressions. The expression **let val** $x = e$ **in** E serves as the bind operator of the monad. Here the bind operator essentially sequences computations – the computations in e are forced to occur before those in E.

It is important to note that in $\lambda_{\bigcirc}$, variables only range over terms and not expressions.

The static semantics of $\lambda_{\bigcirc}$ can be found in Figure 10. We write the judgment $\Gamma \vdash e : \tau$ to mean "term e has type τ with respect to context Γ" and the judgment $\Gamma \vdash E \div \tau$ to mean "expression E has type τ with respect to context Γ". The typing rules for the portion of $\lambda_{\bigcirc}$ corresponding to the simply-typed λ-calculus are standard. The rule WFT:VAL allows for internalizing expression judgments into the term judgments witnessed by the lax modality to denote that the value is a suspended computation. Dually the rule WFE:UNIT allows lifting term judgments to expression judgments, but in this case the injection is not witnessed by a type constructor, because all terms are trivially computations. Finally, the rule WFE:LET shows how **let val** expressions provide a destructor for computations.

Because $\lambda_{\bigcirc}$ has such a different operational character from λ_{FX}, λ_{SEC}, and the simply-typed λ-calculus, we provide its complete dynamics semantics in Figure 11. To be concise, we formalize reduction using call-by-name evaluation contexts. We could have equally well chosen call-by-value. There are three evaluation judgments: $e \rightsquigarrow e'$ for terms making a β-reduction, $E \rightsquigarrow E'$ for expressions making a β-reduction, and $E \rightsquigarrow E'$ for expressions making a reduction under an evaluation context. An expression E is either a finished computation or it can be decomposed into an evaluation context $\mathcal{E}$ and a term or expression with a β-redex. **val** terms are lazy and act as suspensions or thunks, with bind forcing their evaluation.

5.3 Examples

To capture specific notions of computation we can extend the language of $\lambda_{\bigcirc}$ expressions. Consequently, the monadic structure of the lax modality ensures that the bind operator will correctly sequence computations, that is, effects and dependencies.

Mutable references

We can extend $\lambda_{\bigcirc}$ with mutable references by adding expressions for allocating and initializing a reference cell, **ref** e, reading from cells, $!e$, and writing to them, $e_1 := e_2$. The static semantics for these new expressions is straightforward, and given in Figure 12.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}\ e \div \mathsf{ref}\ \tau}\ \textsc{wfe:ref} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash !e \div \tau}\ \textsc{wfe:deref}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 \div \mathsf{ref}\ \tau}\ \textsc{wfe:assn}$$

Figure 12: $\lambda_\bigcirc$'s static semantics extended with mutable references

Heaps	$\mathcal{E}$	::=	$\{l_1 \mapsto v_1, l_2 \mapsto v_2, \ldots\}$
Expression contexts	$\mathcal{E}$	::=	$\bullet \mid [\mathcal{T}] \mid \mathbf{let\ val}\ x = \mathcal{T}\ \mathbf{in}\ E \mid \mathbf{let\ val}\ x = \mathbf{val}\ \mathcal{E}\ \mathbf{in}\ E$
		|	$\mathbf{ref}\ \mathcal{T} \mid !\mathcal{T} \mid \mathcal{T} := e \mid l := \mathcal{T}$

$$\frac{}{(\lambda x{:}\tau.e_1)e_2 \rightsquigarrow\ e_1[e_2/x]}\ \textsc{evb:app}$$

$$\frac{}{\langle H, \mathbf{let\ val}\ x = \mathbf{val}\ [v]\ \mathbf{in}\ E\rangle \rightsquigarrow\ \langle H, E[v/x]\rangle}\ \textsc{evb:letv}$$

$$\frac{}{\langle H, \mathbf{ref}\ v\rangle \rightsquigarrow\ \langle H \cup \{l \mapsto v\}, [l]\rangle}\ \textsc{evb:ref} \qquad \frac{}{\langle H, !l\rangle \rightsquigarrow\ \langle H, [H(l)]\rangle}\ \textsc{evb:deref}$$

$$\frac{}{\langle H, l := v\rangle \rightsquigarrow\ \langle H \cup \{l \mapsto v\}, [l]\rangle}\ \textsc{evb:assn}$$

$$\frac{\langle H, E\rangle \rightsquigarrow\ \langle H', E'\rangle}{\langle H, \mathcal{E}\{E\}\rangle \rightsquigarrow \langle H', \mathcal{E}\{E'\}\rangle}\ \textsc{ev:ctx1} \qquad \frac{e \rightsquigarrow\ e'}{\langle H, \mathcal{E}\{e\}\rangle \rightsquigarrow \langle H', \mathcal{E}\{e'\}\rangle}\ \textsc{ev:ctx2}$$

Figure 13: $\lambda_\bigcirc$'s dynamics semantics revised for mutable references

Adding mutable references to $\lambda_{\bigcirc}$ requires more complicated changes to the dynamics semantics. These are described in Figure 13. Aside from extending the language of expression contexts, we must introduce a notion of a heap, H, a set of mappings between locations, l, and values. We also revise the evaluation judgments for expressions: $\langle H, E\rangle \rightsquigarrow \langle H', E'\rangle$ for expressions making a β-reduction with respect to a heap H, and $\langle H, E\rangle \rightsquigarrow \langle H', E'\rangle$ for expressions making a reduction under an evaluation context with respect to the heap H. It is important to note that the evaluation judgment and reduction rule EVB:APP for terms did not need to change because the strict separation enforced by the lax modality means that terms remain pure and evolve completely independently of the heap.

Our simple example of memory allocation from Section 3 naïvely becomes

$$\cdot \vdash \lambda x{:}\mathsf{int}.(\lambda y{:}\bigcirc\mathsf{ref\ int}.0)(\mathbf{val\ ref}\ x) : \mathsf{int} \to \mathsf{int} \tag{9}$$

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

$$\cdot \vdash \lambda x{:}\mathsf{int}.\mathbf{val}\ (\mathbf{let\ val}\ z = (\mathbf{val\ ref}\ x)\ \mathbf{in}\ [(\lambda y{:}\mathsf{ref\ int}.0)z]) : \mathsf{int} \to \bigcirc\mathsf{int} \tag{10}$$

This revised version correctly captures the intended behavior, and now the lax modality in the function's type reflects that the result is actually a computation.

Secure information-flow

Interestingly, the sequencing of the monad can also be used to track dependencies. Consider our example from Section 4, where we wish to track which parts of a program may depend upon high-security information. We can again introduce a high security constant for a password.

$$\frac{}{\Gamma \vdash \mathbf{passwd} \div \mathsf{int}}\ \text{WFE:PW}$$

Here instead of giving the password a high-security label, we simply add **passwd** to the language of expressions. For any piece of code to actually make use of **passwd**, it must first bind it, as such

$$\cdot \vdash \mathbf{let\ val}\ x = \mathbf{passwd\ in}\ \ldots \div \tau \tag{11}$$

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

We can even state an analogous noninterference theorem for $\lambda_{\bigcirc}$

Theorem 5.2 (Noninterference)**.** *If* $\cdot, x : \bigcirc\mathsf{int} \vdash e : \mathsf{int}$ *then for any two* $\cdot \vdash e_1 : \bigcirc\mathsf{int}$ *and* $\cdot \vdash e_2 : \bigcirc\mathsf{int}$ *the result of computing* $e[e_1/x]$ *and* $e[e_2/x]$ *will be equivalent.*

Proof. The canonical proof is a corollary of a more general proof of substitution for a logical relation, but syntactic techniques also exist. □

Nontermination

Finally, we can also track nontermination in $\lambda_{\bigcirc}$ by making the fix-point operator an expression, such that any potentially diverging computation must be explicitly forced by using bind. However, the design of this fix-point takes some care.

$$\frac{\Gamma, x{:}\bigcirc\tau \vdash E \div \tau}{\Gamma \vdash \mathbf{fix}\ x{:}\tau.E \div \tau}\ \textsc{wfe:fix} \qquad \frac{}{\mathbf{fix}\ x{:}\tau.E \rightsquigarrow E[\mathbf{val}\ (\mathbf{fix}\ x{:}\tau.E)/x]}\ \textsc{evb:fix}$$

Because the fix-point evaluates by unwinding to its body, and because the fix-point is an expression itself, we require that the body be an expression too. However, because variables in $\lambda_{\bigcirc}$ only range over terms, when substituting itself in its body, the fix-point must first wrap itself inside a **val**.

Our running example of factorial would become

$$\cdot \vdash \mathbf{fix}\ \mathtt{fact}{:}\bigcirc(\mathsf{int} \to \bigcirc\mathsf{int}). \qquad \div \mathsf{int} \to \bigcirc\mathsf{int} \quad (12)$$
$$[\lambda x{:}\mathsf{int}.\mathbf{if}\ x \leq 0\ \mathbf{then}$$
$$\mathbf{val}\ [1]$$
$$\mathbf{else}$$
$$\mathbf{val}\ (\mathbf{let\ val}\ \mathtt{fact}' = \mathtt{fact}\ \mathbf{in}\ [x * \mathtt{fact}'(x-1)])]$$

As with λ_{FX} and λ_{SEC} this type is quite conservative. Because of the use of a fix-point to compute the factorial function, the type of function itself is marked as a potentially diverging computation. Similarly, even though the result of computing factorial will always be a value, the return type must be conservatively declared an integer computation to indicate that the function could diverge.

$$\text{Types} \quad \tau \quad ::= \quad \ldots \mid \bigcirc_\ell \tau \mid \ldots$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{wft:sub}$$

$$\frac{\Gamma \vdash E \div_\ell \tau}{\Gamma \vdash \mathbf{val}\ E : \bigcirc_\ell \tau}\ \textsc{wft:vall} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] \div_\varnothing \tau}\ \textsc{wfe:unitl}$$

$$\frac{\Gamma \vdash e_1 : \bigcirc_{\ell_1} \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 \div_{\ell_2} \tau_2 \qquad \ell_1 \leq \ell_2}{\Gamma \vdash \mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \div_{\ell_2} \tau_2}\ \textsc{wfe:letl}$$

$$\frac{\Gamma \vdash E \div_{\ell_1} \tau \qquad \ell_1 \leq \ell_2}{\Gamma \vdash E \div_{\ell_2} \tau}\ \textsc{wfe:sub}$$

Figure 14: Labeled semantics for $\lambda_{\bigcirc}$.

$$\frac{}{\tau \leq \tau}\ \textsc{sub:refl} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \textsc{sub:trans}$$

$$\frac{\tau_1 \leq \tau_2 \qquad \ell_1 \leq \ell_2}{\bigcirc_{\ell_1} \tau_1 \leq \bigcirc_{\ell_2} \tau_2}\ \textsc{sub:lax} \qquad \frac{\tau_3 \leq \tau_1 \qquad \tau_2 \leq \tau_4}{(\tau_1 \to \tau_2) \leq (\tau_3 \to \tau_4)}\ \textsc{sub:arr}$$

Figure 15: The subtyping relation for labeled $\lambda_{\bigcirc}$.

5.4 Labelled $\lambda_{\bigcirc}$

In the previous section we saw that the lax modality provides an elegant approach to modelling effects and dependencies. However, it fails to be as descriptive as the labels we used in λ_{FX} and λ_{SEC}. A term with type $\bigcirc\tau$ is some computation producing a value τ, but the nature of the computation is completely hidden. For example, our simple function that performs a vacuous allocation

$$\cdot \vdash \lambda x{:}\mathsf{int}.\mathbf{val}\ (\mathbf{let\ val}\ z = (\mathbf{val\ ref}\ x)\ \mathbf{in}\ [(\lambda y{:}\mathsf{ref\ int}.0)z]) : \mathsf{int} \to \bigcirc\mathsf{int} \tag{13}$$

indicates by its monadic type that it is not pure, but it does not tell us it was because of allocation, rather than reading or writing to a mutable reference.

$$
\begin{array}{rcl}
[\![\mathsf{int}]\!] & \triangleq & \mathsf{int} \\
[\![\tau_1 \xrightarrow{\ell} \tau_2]\!] & \triangleq & [\![\tau_1]\!] \to \bigcirc_\ell [\![\tau_2]\!] \\
\\
[\![i]\!] & \triangleq & i \\
[\![\lambda x{:}\tau.e]\!] & \triangleq & \lambda x{:}[\![\tau]\!].\mathbf{val}\ [\![e]\!] \\
\\
[\![v]\!] & \triangleq & [[\![v]\!]^*] \\
[\![x]\!] & \triangleq & [x] \\
[\![e_1 e_2]\!] & \triangleq & \mathbf{let\ val}\ x_1 = (\mathbf{val}\ [\![e_1]\!])\ \mathbf{in} \\
 & & \quad \mathbf{let\ val}\ x_2 = (\mathbf{val}\ [\![e_2]\!])\ \mathbf{in}\ (\mathbf{let\ val}\ x = x_1 x_2\ \mathbf{in}\ [x]) \\
\\
[\![\cdot]\!] & \triangleq & \cdot \\
[\![\Gamma, x{:}\tau]\!] & \triangleq & [\![\Gamma]\!], x{:}[\![\tau]\!] \\
\\
[\![\Gamma \vdash_\ell e : \tau]\!] & \triangleq & [\![\Gamma]\!] \vdash [\![e]\!] \div_\ell [\![\tau]\!]
\end{array}
$$

Figure 16: Encoding of λ_{FX} into labelled $\lambda_{\bigcirc}$

We can obtain the same level of precision as in λ_{FX} and λ_{SEC} if we can extend $\lambda_{\bigcirc}$ so that it has not just one monadic modality, but an entire lattice of them. This approach has been taken by both Wadler [Wad98] and Abadi et. al. [ABHR99] for effects and dependencies respectively. In Figures 10 and 15 we give the grammar, static semantics, and induced subtyping relation for such an extension.

The new typing rule WFE:UNITL for the monadic unit reflects the fact that because values in labelled $\lambda_{\bigcirc}$ are pure, they can be labelled with the least dependency or effect in the lattice. The typing rule WFE:LETL can be seen as capturing the requirement that effects and dependencies are propagated during evaluation.

To illustrate the expressiveness of the labelled version of $\lambda_{\bigcirc}$, we show that it can soundly encode both λ_{FX} and λ_{SEC}. We stop short of showing that the encodings bisimulate the source languages for reasons of space.

Encoding λ_{FX} into labelled $\lambda_{\bigcirc}$

Our encoding of λ_{FX} into labelled $\lambda_{\bigcirc}$, shown in Figure 16, is inspired by Wadler [Wad98], but his target language did not make the same distinction between terms and expressions, so it has been adapted some. However, the type encoding remains exactly the same. Functions types in λ_{FX} are encoded as pure functions in $\lambda_{\bigcirc}$ that return computations.

Our encoding makes a distinction between encoding values and terms. The operator $[\![\cdot]\!]^*$ maps λ_{FX} values to $\lambda_{\bigcirc}$ terms, and the operator $[\![\cdot]\!]$ maps λ_{FX} terms to $\lambda_{\bigcirc}$ expressions. This captures the intuition that in λ_{FX} values are inert, just as terms in $\lambda_{\bigcirc}$ are pure, and because λ_{FX} terms may produce effects they should be mapped to expressions in $\lambda_{\bigcirc}$, which isolate computations.

Regardless of whether we take $\lambda_{\bigcirc}$ to be call-by-value or call-by-name, the above encoding provides the correct call-by-value operational behavior as the sequencing provided by the **let val** expressions in the encoding of applications ensures that both the function and argument are evaluated before they are applied. The final nested **let val** forces the computation produced by all encoded functions.

Theorem 5.3 (Encoding of λ_{FX} sound)**.**

1. *If* $\tau_1 \leq \tau_2$ *then* $[\![\tau_1]\!] \leq [\![\tau_2]\!]$.
2. *If* $\Gamma \vdash_{\varnothing} v : \tau$ *then* $\Gamma \vdash [\![v]\!]^* : [\![\tau]\!]$.
3. *If* $\Gamma \vdash_{\ell} e : \tau$ *then* $\Gamma \vdash [\![e]\!] \div_{\ell} [\![\tau]\!]$.

Proof. Part 1 follows from straightforward induction over the subtyping derivation. Parts 2 and 3 follow from mutual induction over the structure of the typing derivation. □

Encoding λ_{SEC} *into labelled* $\lambda_{\bigcirc}$

Our encoding of λ_{SEC} into labelled $\lambda_{\bigcirc}$ is inspired by Abadi et. al.'s [ABHR99] encoding of languages for tracking dependencies into a monadic calculus. Again, their target language did not have an explicit separation between terms and expressions, so there are some differences.

The encoding is in some respects almost a dual of the encoding used for λ_{FX}, but we will not attempt to formalize the duality here. In the encoding described in Figure 17, λ_{SEC} values are encoded as $\lambda_{\bigcirc}$ expressions, and λ_{SEC} terms are encoded as $\lambda_{\bigcirc}$ terms. This is because it is the values in λ_{SEC} that carry the dependencies, and to capture this their encoding must live in the world of $\lambda_{\bigcirc}$ computations. Despite encoding λ_{SEC} terms into $\lambda_{\bigcirc}$ terms, we must encode λ_{SEC} programs into a $\lambda_{\bigcirc}$ expression, because expressions form the basis of evaluation in $\lambda_{\bigcirc}$. If we simply encoded λ_{SEC} programs as $\lambda_{\bigcirc}$ terms, the resulting terms would be always be values and there would be no correspondence in evaluations.

The encoding above will only suffice for the call-by-name version of λ_{SEC}, even if we were to give $\lambda_{\bigcirc}$ a call-by-value semantics. For the encoding to produce a call-by-value semantics, the case for application must be changed to the following

$$\begin{aligned}[\![e_1 e_2]\!] \triangleq\ & \mathbf{val}\ (\mathbf{let\ val}\ x_1 = [\![e_1]\!]\ \mathbf{in} \\ & \quad \mathbf{let\ val}\ x_2 = [\![e_2]\!]\ \mathbf{in}\ (\mathbf{let\ val}\ x = x_1(\mathbf{val}\ [x_2])\ \mathbf{in}\ [x]))\end{aligned}$$

$$
\begin{array}{rcl}
[\![\mathsf{int}_\ell]\!] & \triangleq & \bigcirc_\ell \mathsf{int} \\
[\![\tau_1 \to_\ell \tau_2]\!] & \triangleq & \bigcirc_\ell([\![\tau_1]\!] \to [\![\tau_2]\!]) \\
\\
[\![i]\!]^* & \triangleq & [i] \\
[\![\lambda x{:}\tau.e]\!]^* & \triangleq & [\lambda x{:}[\![\tau]\!].[\![e]\!]] \\
\\
[\![v]\!] & \triangleq & \mathbf{val}\ [\![v]\!]^* \\
[\![x]\!] & \triangleq & x \\
[\![e_1 e_2]\!] & \triangleq & \mathbf{val}\ (\mathbf{let\ val}\ x_1 = [\![e_1]\!]\ \mathbf{in}\ (\mathbf{let\ val}\ x_2 = x_1 [\![e_2]\!]\ \mathbf{in}\ [x_2])) \\
\\
[\![\cdot]\!] & \triangleq & \cdot \\
[\![\Gamma, x{:}\tau]\!] & \triangleq & [\![\Gamma]\!], x{:}[\![\tau]\!] \\
\\
[\![\Gamma \vdash e : \tau]\!] & \triangleq & [\![\Gamma]\!] \vdash \mathbf{let\ val}\ x = [\![e]\!]\ \mathbf{in}\ [x] \div_\ell \tau' \\
 & & \text{where } [\![\tau]\!] = \bigcirc_\ell \tau'
\end{array}
$$

Figure 17: Encoding of λ_{SEC} into $\lambda_\bigcirc$

Here the additional **let val** is used to force the evaluation of the function argument, before it is packaged back up again using **val** and the monadic unit. This revised encoding will produce the call-by-value semantics regardless of whether the semantics of $\lambda_\bigcirc$ are call-by-name or call-by-value.

Theorem 5.4 (Encoding of λ_{SEC} sound)**.**

1. *If* $\tau_1 \leq \tau_2$ *then* $[\![\tau_1]\!] \leq [\![\tau_2]\!]$.
2. *If* $\Gamma \vdash v : \tau$ *then* $\Gamma \vdash [\![v]\!]^* \div_\ell \tau'$ *where* $[\![\tau]\!] = \bigcirc_\ell \tau'$.
3. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash [\![e]\!] : [\![\tau]\!]$.

Proof. Part 1 follows from straightforward induction over the subtyping derivation. Parts 2 and 3 follow from mutual induction over the structure of the typing derivation. □

6 Conclusion

The type systems for reasoning and effects and dependencies presented in this paper provide an important foundation for reasoning about realistic programs. We can objectively say that the use of monadic structures to encapsulate effects has revolutionized functional programming.

Still, the survey in this paper has only scratched surface. Another important paradigm for formalizing the effectful behavior of programs are substructural type systems. Much of the interest in substructural type systems evolved out of research into linear logic. Linear logic was originally conceived from research into domain theory and coherence semantics [GLT89]. 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.

Furthermore, there appears to be a strong connection between monadic type systems and linear type systems. Benton and Wadler developed a language, the adjoint calculus [BW96], that cleanly combines a monadic and linear computations such that they are delightfully close to duals [BW96]. However, the adjoint calculus only models monadic structures that are commutative. Pereira has conjectured [Per05] that their may be a better fit in attempting to relate monadic languages with ordered substructural logics such as the Lambek calculus [Lam58] and Polakow's ordered linear logic [Pol01].

Recently, it has also been recognized that monads might not be the most fundamental structure for encapsulating effects and dependencies. In Pfenning and Davies' judgmental reconstruction of modal logic [PD01], they were able to show that the lax modality can be cleanly decomposed into a combination of modal possibility and modal necessity. The fact that the lax modality seems to have properties of both necessity and possibility had already been noted by Fairtlough and Mendler [FM97]. What makes this result even more fascinating is that modal possibility and necessity are duals, and while modal possibility itself has a monadic structure, modal necessity forms a *co-monadic* structure. 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.

Indeed, it has actually been suggested previously that a co-monadic structure might be more appropriate for representing effects that arise from the context in which a program fragment may execute [Kie99, Par00, Nan04, PH04]. Kieburtz was the first to propose the use of co-monads, 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 [GP99] and the judgmental 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 [Nan04].

Because of the dichotomy between possibility and necessity, monadic and comonadic, we believe that there exists more refined encodings of λ_{FX} and λ_{SEC} into a language with modal necessity and modal possibility than our monadic encodings into $\lambda_{\bigcirc}$. In passing we have noted the seemingly dual nature of effects and dependencies. We

conjecture that this duality will manifest itself as a natural encodings of dependencies using modal necessity and effects using modal possibility. However, formalizing this duality precisely will require further study. We believe that by making this duality explicit in type systems it will provide a more unified account of program behavior and provide additional expressive power.

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 helpful in pointing me in the correct direction for some 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, even though that material was cut from the final draft. I am also thankful for my advisor Stephanie Weirich.

Colophon

This document prepared using the LaTeX typesetting system created by Leslie Lamport with the memoir class. The document was processed using pdfTeX's microtypography extensions implemented by Hàn Thế Thành. The body text is set at 11pt. The serif typefaces are from the Minion Pro Opticals family, designed by Robert Slimbach of Adobe Systems. The sans-serif typefaces are from the Cronos Pro Opticals family, also designed by Robert Slimbach. The monospace typeface is Adobe Letter Gothic designed by Roger Roberson. The serif mathematics typeface is ams Euler, designed by Hermann Zapf. Some mathematical symbols are from Gentzen Symbol, designed by myself.

Bibliography

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon 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, January 1999.

[AM97] Samson Abramsky and Guy McCusker. Game semantics. In *Proceedings of Marktorberdorf '97 Summerschool*, 1997. Lecture notes.

[Bac81] John Backus. The history of Fortran I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[BL75] David E. Bell and Len J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975.

[BW96] Nick Benton and Phil Wadler. Linear logic, monads and the lambda calculus. In *Proceedings of the Eleventh IEEE Symposium on Logic in Computer Science*, Brunswick, New Jersey, July 1996. IEEE Press.

[Con90] Charles Consel. Binding time analysis for high order untyped functional languages. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 264–272. ACM Press, 1990.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the Eleventh IEEE Symposium on Logic in Computer Science*, pages 184–195, Brunswick, New Jersey, July 1996. IEEE Press.

[ECM02] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002. Also ISO/IEC 23270.

[FM97] M. V. H. Fairtlough and M. Mendler. Propositional Lax Logic. *Information and Computation*, 137(1):1–33, August 1997.

[GJLS87] David Gifford, Pierre Jouvelot, John Lucassen, and Mark Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science*, pages 214–224, Trento, Italy, July 1999. IEEE Press.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[Kie99] Richard Kieburtz. Codata and comonads in haskell. Available as a draft from `ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/haskell99.ps`, July 1999.

[Kri63] Saul A. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Nan04] Aleksander Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, Carnegie Mellon University, August 2004. Carnegie Mellon Technical Report CMU-CS-04-151.

[PAC94] Gordon Plotkin, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 310–319, 1994.

[Par00] Alberto Pardo. Towards merging recursion and comonads. Technical Report UU-CS-2000-19, Utrecht University, July 2000.

[PD01] Frank Pfenning and Rowan Davies. A judgemental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[Per05] Fernando C. N. Pereira. Personal communication, February 2005.

[PH04] Sungwoo Park and Robert Harper. A logical view of effects. Available as a draft from `http://www-2.cs.cmu.edu/~rwh/papers/modaleff/short.pdf`, 2004.

[PHA+99] 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/`, February 1999.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, August 2001. Carnegie Mellon Technical Report CMU-CS-01-152.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effects discipline. In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 162–173, Santa Cruz, CA, June 1992. IEEE Press.

[Wad89] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. Imperial College, London.

[Wad92] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, volume 2, pages 461–493. Cambridge University Press, 1992.

[Wad98] Philip Wadler. The marriage of effects and monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional programming*, pages 63–74, Baltimore, Maryland, September 1998. ACM Press.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Zda02] Stephan Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.